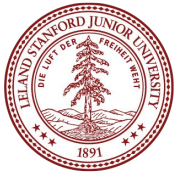
Divide-and-Conquer Sorting Algorithms

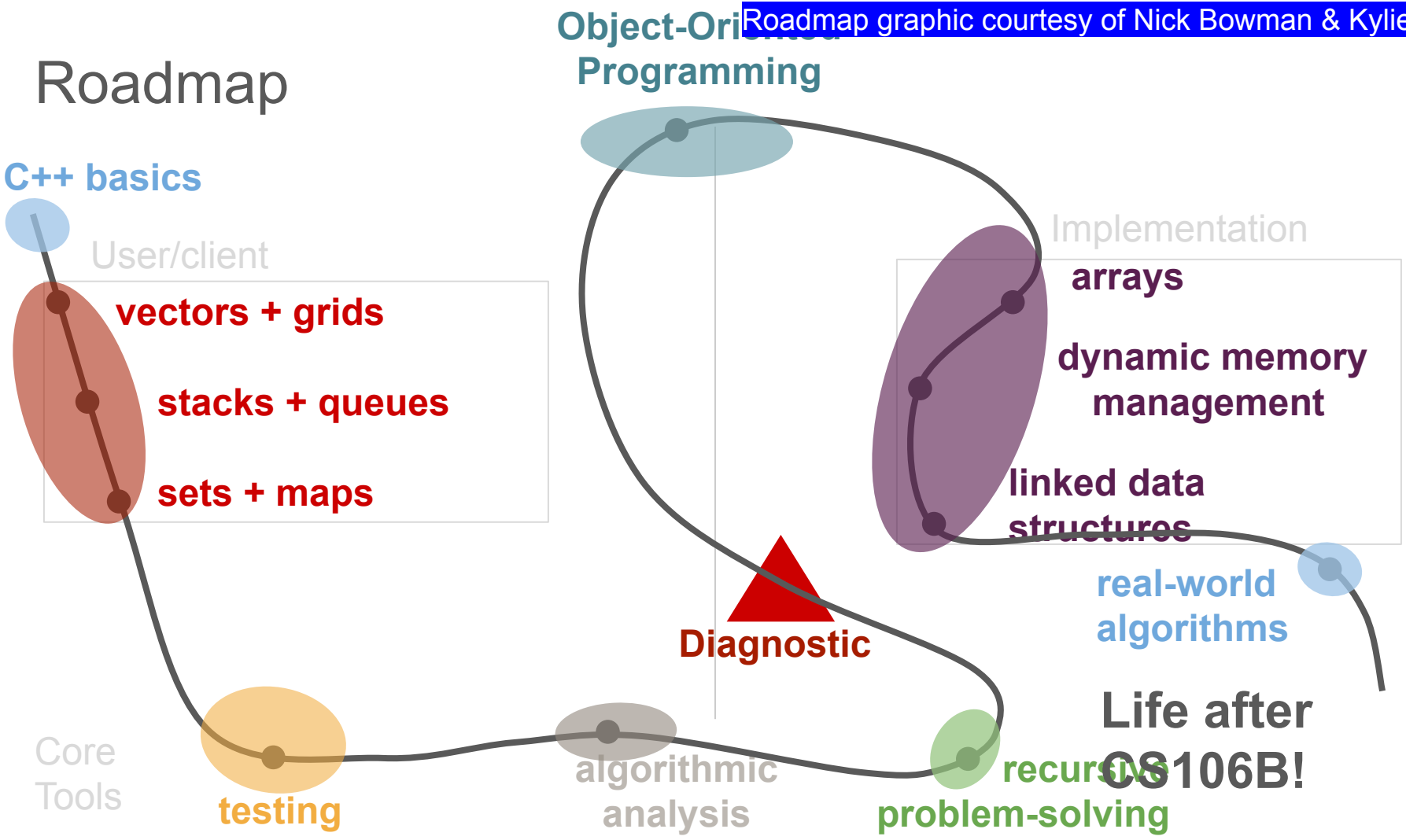
What is an example of a real-world problem that's made easier by dividing it up across many people?

(put your answers in the chat)

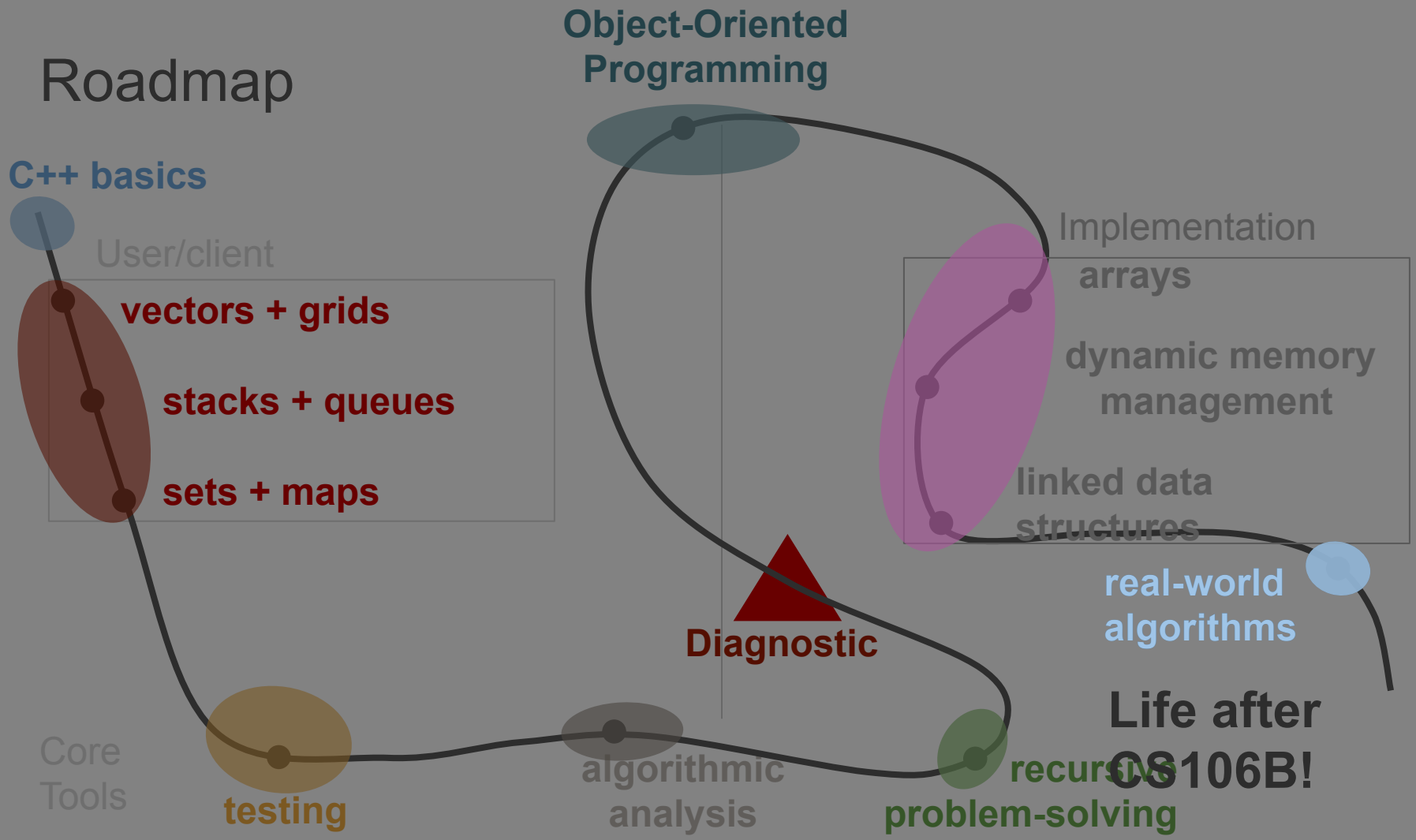


Roadmap graphic courtesy of Nick Bowman & Kylie Jue

Roadmap



Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Today's questions

How can we design better, more efficient sorting algorithms?

Today's topics

1. Review
2. Merge Sort
3. Quicksort

Review

[linked list wrapup + intro to sorting]

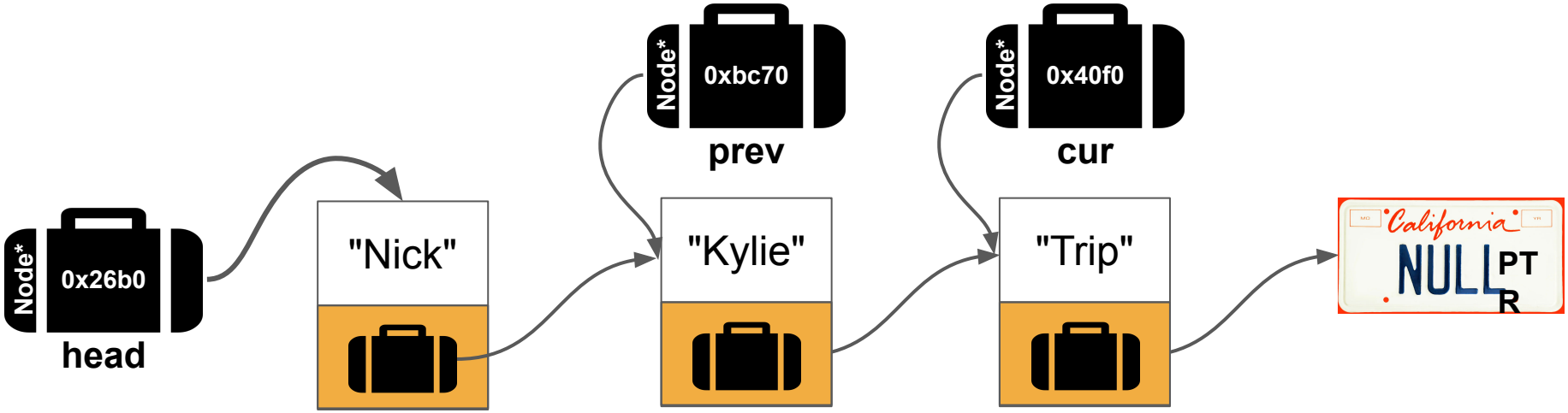
Linked List Wrapup

Common linked lists operations

- **Traversal**
  - How do we walk through all elements in the linked list?
- **Rewiring**
  - How do we rearrange the elements in a linked list?
- **Insertion**
  - How do we add an element to a linked list?
- **Deletion**
  - How do we remove an element from a linked list?

Takeaways for manipulating the middle of a list

- While traversing to where you want to add/remove a node, you'll often want to keep track of both a current pointer and a previous pointer.
  - This makes rewiring easier between the two!
  - This also means you have to check that neither is nullptr before dereferencing.



Linked list summary

- You've now learned lots of ways to manipulate linked lists!
  - Traversal
  - Rewiring
  - Insertion (front/back/middle)
  - Deletion (front/back/middle)

- You've seen linked lists in classes and outside classes, and pointers passed by value and passed by reference.

- Assignment 5 will really test your understanding of linked lists.
  - Draw lots of pictures!
  - Test small parts of your code at a time to make sure individual operations are working correctly.

Sorting

Definition

sorting

Given a list of data points, sort those data points into ascending / descending order by some quantity.

Selection sort takeaways

- Selection sort works by "selecting" the smallest remaining element in the list and putting it in the front of all remaining elements.

- Selection sort is an $O(n^2)$ algorithm.

Insertion sort algorithm

- Repeatedly insert an element into a sorted sequence at the front of the array.

- To insert an element, swap it backwards until either:
  - (1) it’s bigger than the element before it, or
  - (2) it’s at the front of the array.

The complexity of insertion sort

- In the worst case (the array is in reverse sorted order), insertion sort takes time $O(n^2)$.
  - The analysis for this is similar to selection sort!

- In the best case (the array is already sorted), insertion takes time O(n) because you only iterate through once to check each element.
  - Selection sort, however, is always $O(n^2)$ because you always have to search the remainder of the list to guarantee that you're finding the minimum at each step.

- **Fun fact**: Insertion sorting an array of random values takes, *on average*, $O(n^2)$ time.
  - This is beyond the scope of the class – take an advanced statistics or algorithms class if you're interested in learning more!

Let's do better than $O(N^2)$ sorting!

Advanced Sorting

How can we design better, more efficient sorting algorithms?

Divide-and-Conquer

Motivating Divide-and-Conquer

- So far, we've seen $O(N^2)$ sorting algorithms. How can we start to do better?

Motivating Divide-and-Conquer

- So far, we've seen $O(N^2)$ sorting algorithms. How can we start to do better?

- Assume that it takes t seconds to run insertion sort on the following array:

14	6	3	9	7	16	2	15	5	10	8	11	1	13	12	4

Motivating Divide-and-Conquer

- So far, we've seen $O(N^2)$ sorting algorithms. How can we start to do better?
- Assume that it takes `t` seconds to run insertion sort on the following array:

14	6	3	9	7	16	2	15	5	10	8	11	1	13	12	4

- Poll: Approximately how many seconds will it take to run insertion sort on **each** of the following arrays?

14	6	3	9	7	16	2	15

5	10	8	11	1	13	12	4

Motivating Divide-and-Conquer

- So far, we've seen $O(N^2)$ sorting algorithms. How can we start to do better?
- Assume that it take… the following array:

Answer: Each array should only take about t/4 seconds to sort.

- Poll: Approximately … un insertion sort on **each** of the following arrays?

Motivating Divide-and-Conquer

- Main insight:
  - Sorting `N` elements directly takes total time `t`
  - Sorting two sets of `N/2` elements (total of `N` elements) takes total time `t/2`
  - We got a speedup just by sorting smaller sets of elements at a time!

Motivating Divide-and-Conquer

- Main insight:
  - Sorting `N` elements directly takes total time `t`
  - Sorting two sets of `N/2` elements (total of `N` elements) takes total time `t/2`
  - We got a speedup just by sorting smaller sets of elements at a time!

- The main idea behind divide-and-conquer algorithms takes advantage of this. Let's design algorithms that break up a problem into many smaller problems that can be solved in parallel!

General Divide-and-Conquer Approach

- Our general approach when designing a divide-and-conquer algorithm is to decide how to make the problem smaller and how to unify the results of these solved, smaller problems.

General Divide-and-Conquer Approach

- Our general approach when designing a divide-and-conquer algorithm is to decide how to make the problem smaller and how to unify the results of these solved, smaller problems.

- Both sorting algorithms we explore today will have both of these components:
  - Divide Step
    - Make the problem smaller by splitting up the input list
  - Join Step
    - Unify the newly sorted sublists to build up the overall sorted result

General Divide-and-Conquer Approach

- Our general approach when designing a divide-and-conquer algorithm is to decide how to make the problem smaller and how to unify the results of these solved, smaller problems.

- Both sorting algorithms we explore today will have both of these components:
  - Divide Step
    - Make the problem smaller by splitting up the input list
  - Join Step
    - Unify the newly sorted sublists to build up the overall sorted result

- Divide-and-Conquer is a ripe time to return to recursion!

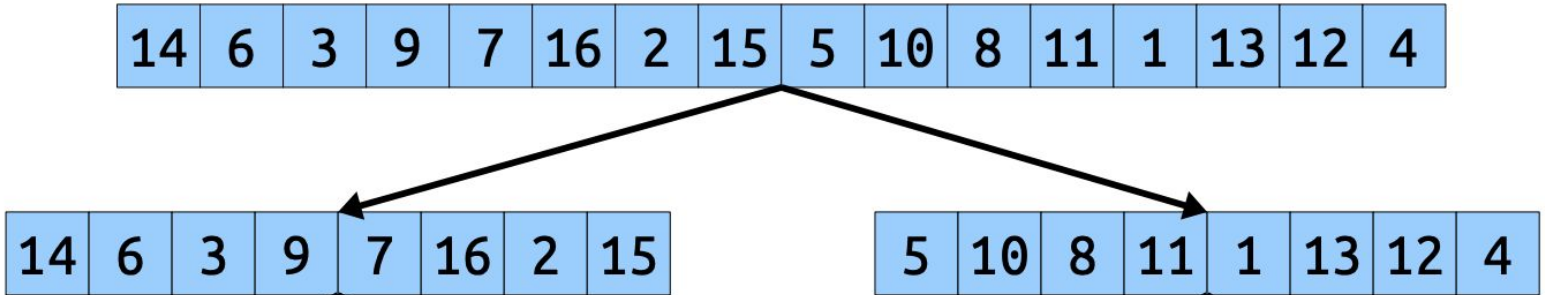
Merge Sort

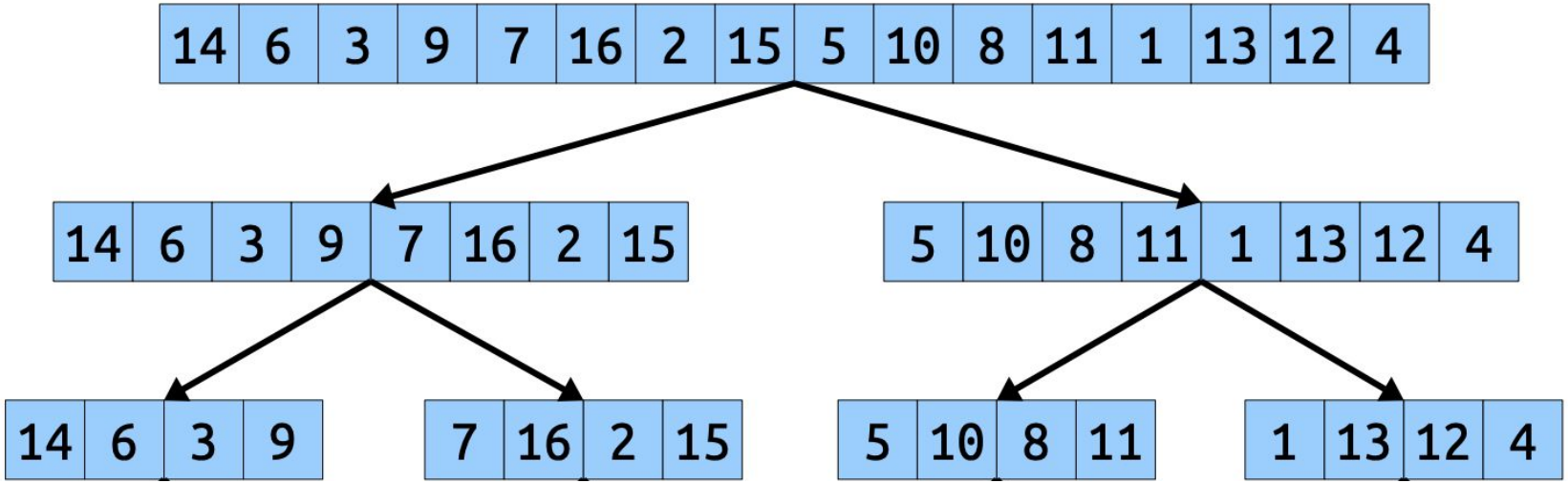
Merge Sort

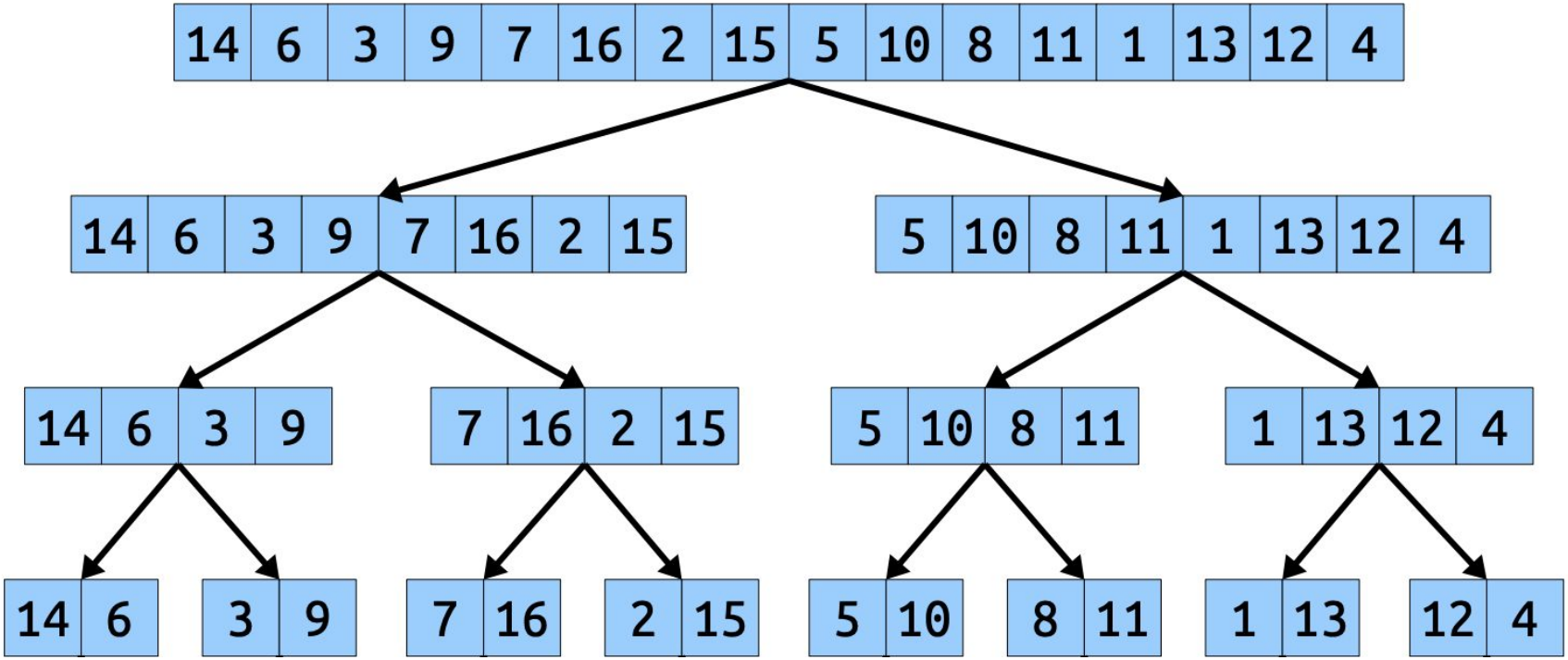
A recursive sorting algorithm!

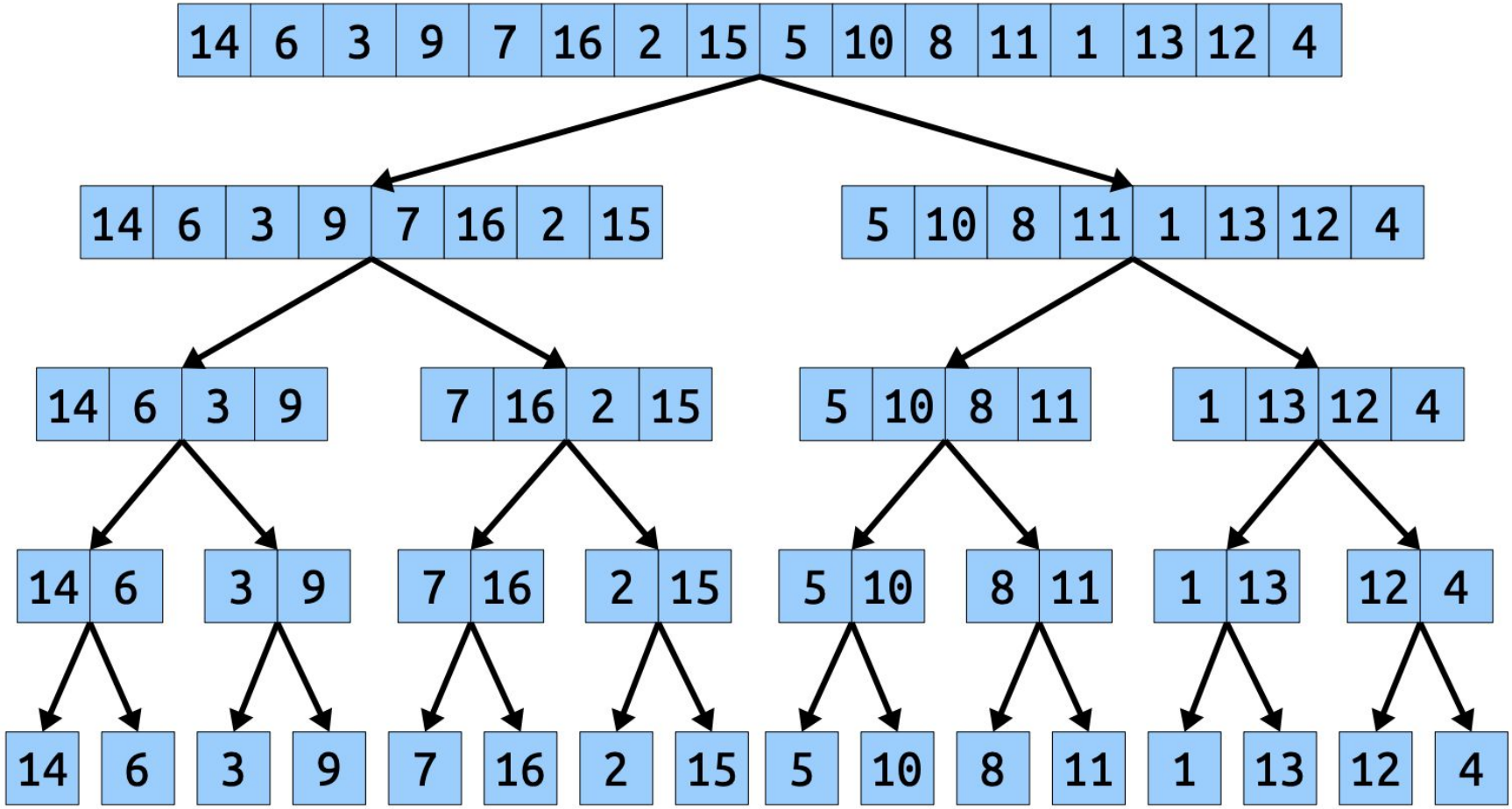
- **Base Case:**
  - An empty or single-element list is already sorted.
- **Recursive step:**
  - Break the list in half and recursively sort each part. (easy divide)
  - Use merge to combine them back into a single sorted list (hard join)

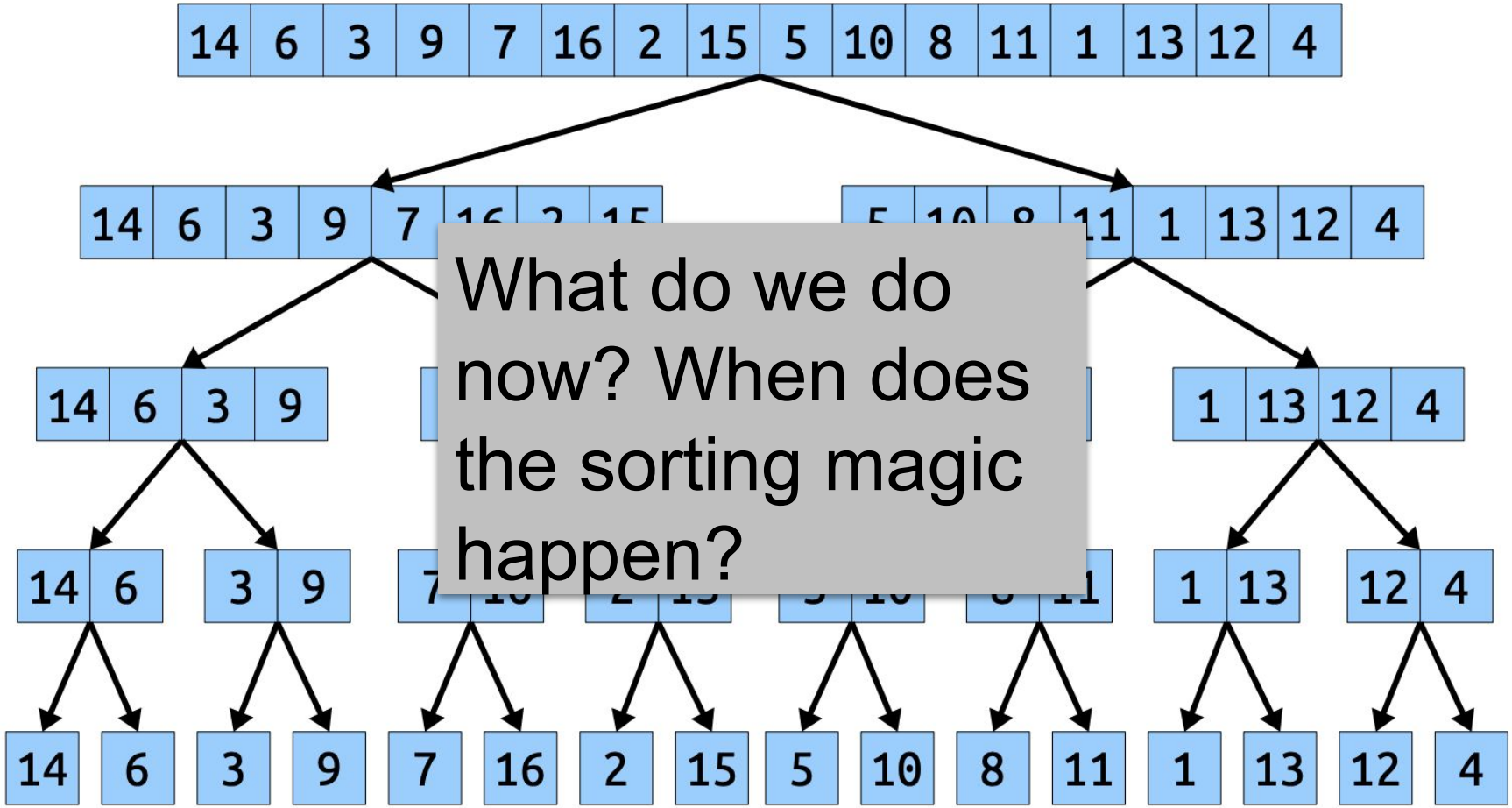

14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4


14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4


14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4

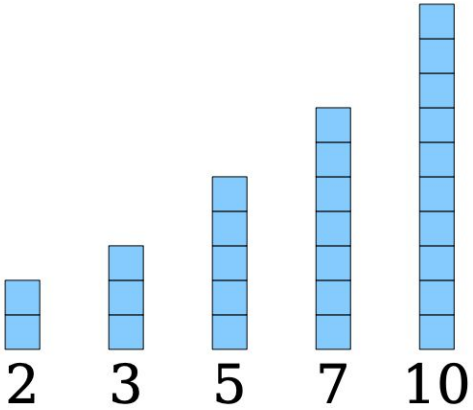

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4


14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

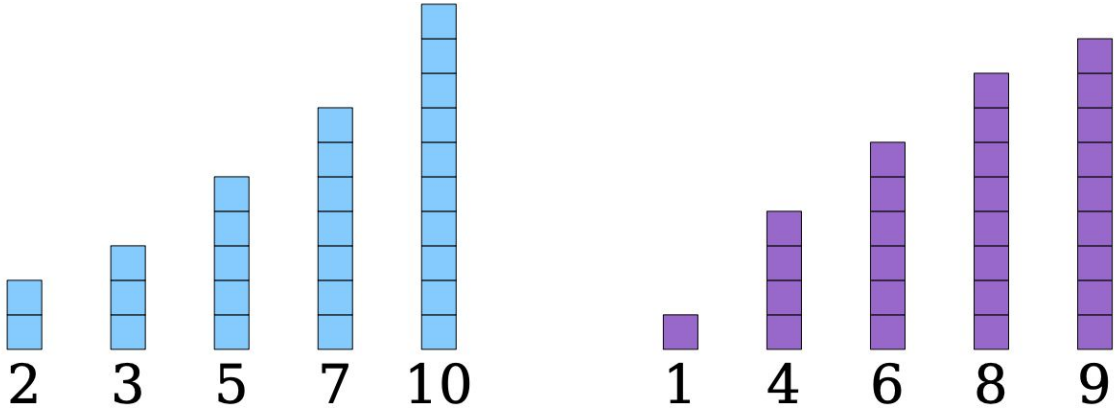

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
1 13 12 4
14 6
3 9
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4
What do we do now? When does the sorting magic happen?

The Key Insight: Merge

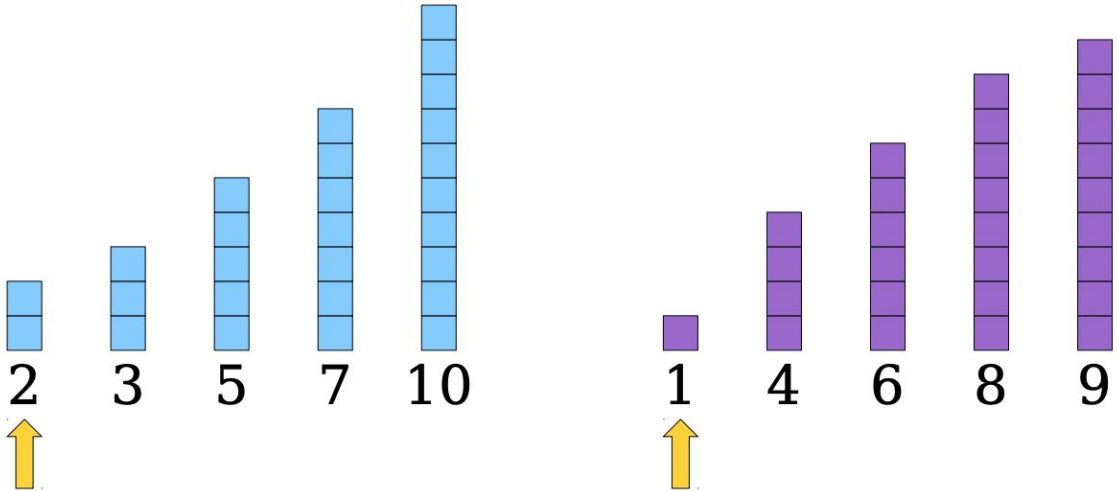
The Key Insight: Merge



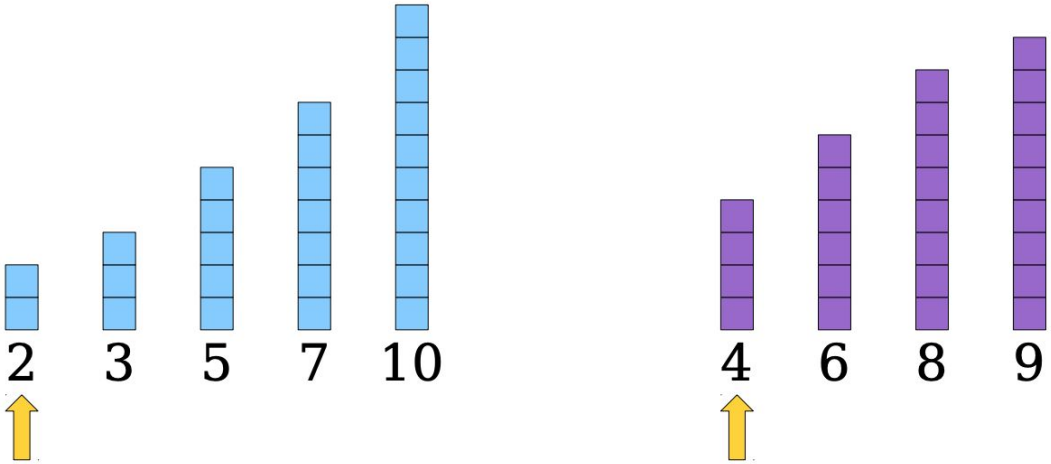
The Key Insight: Merge



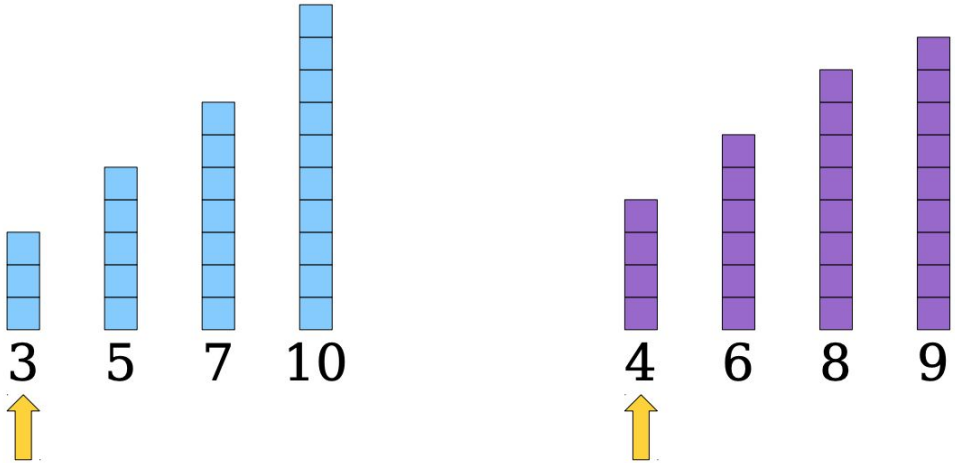
The Key Insight: Merge

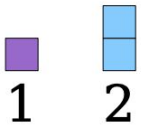


The Key Insight: Merge

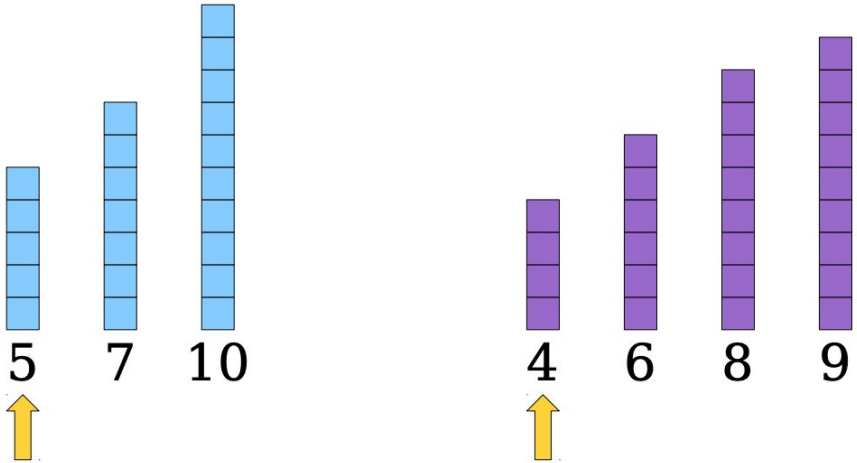


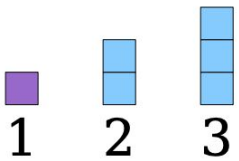
The Key Insight: Merge



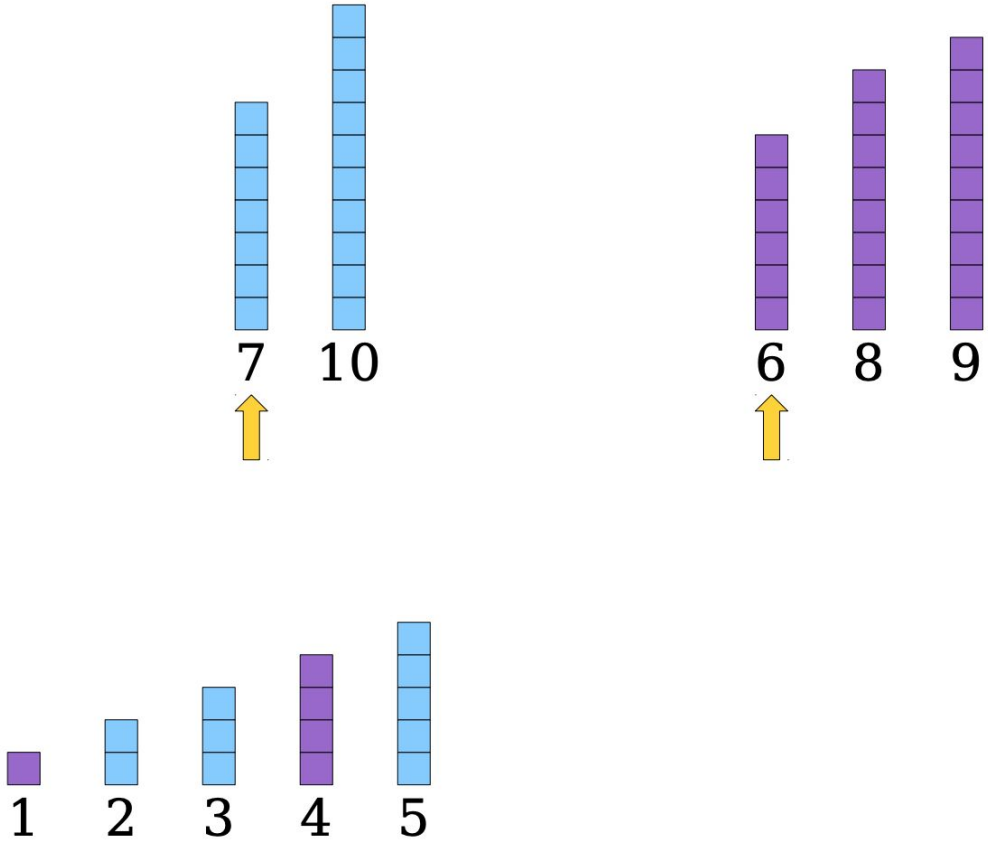


The Key Insight: Merge

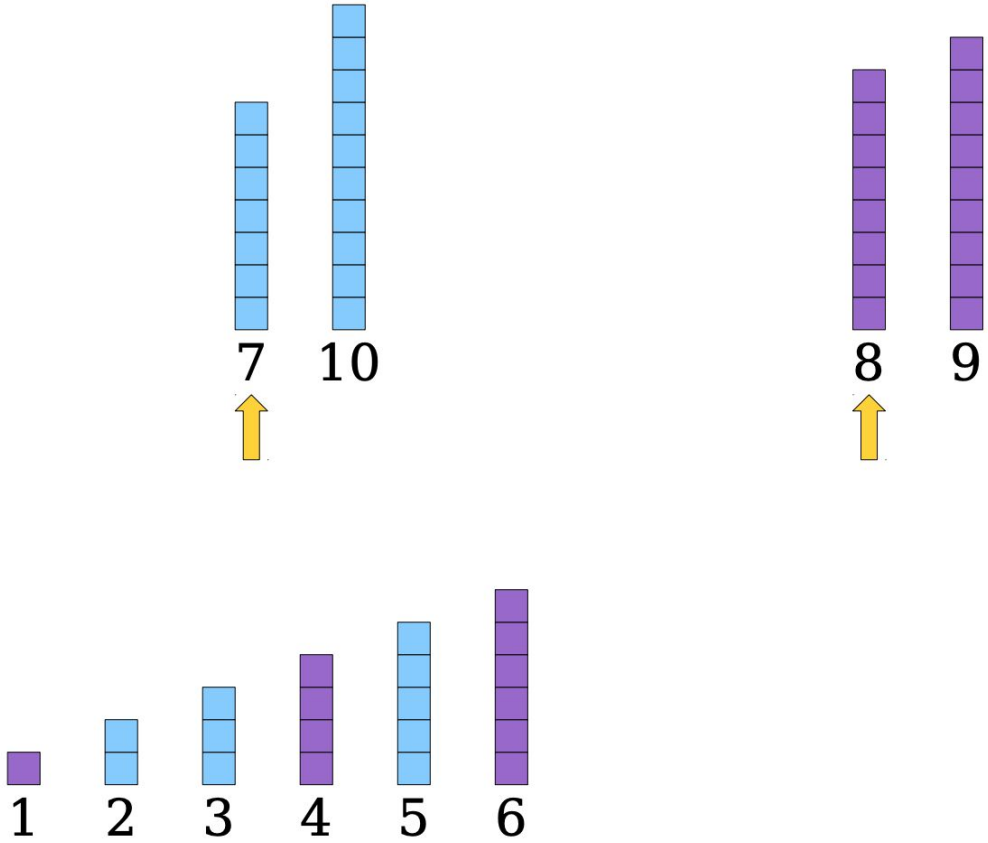




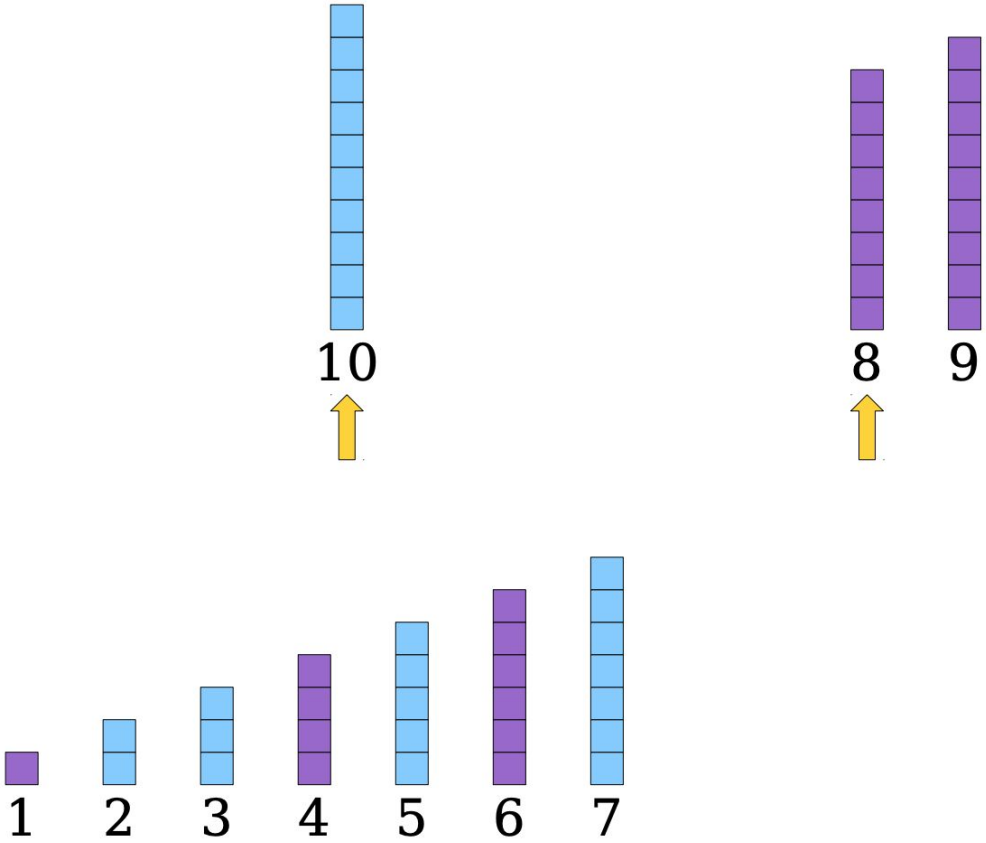
The Key Insight: Merge



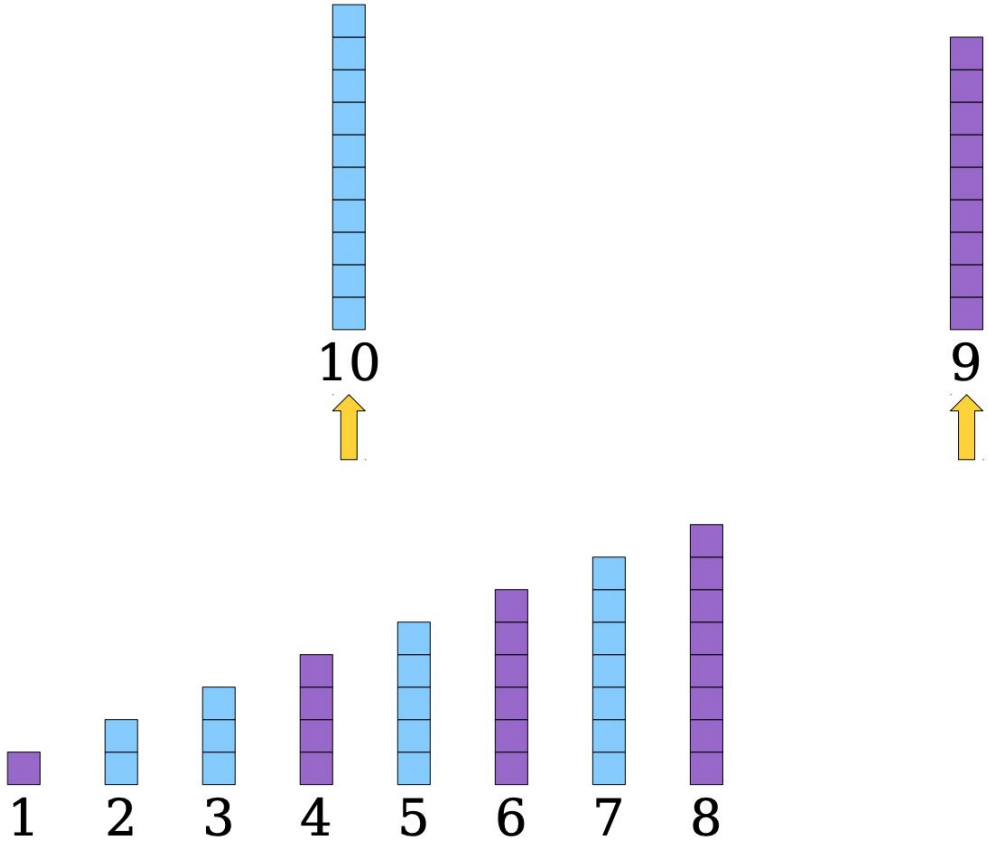
The Key Insight: Merge



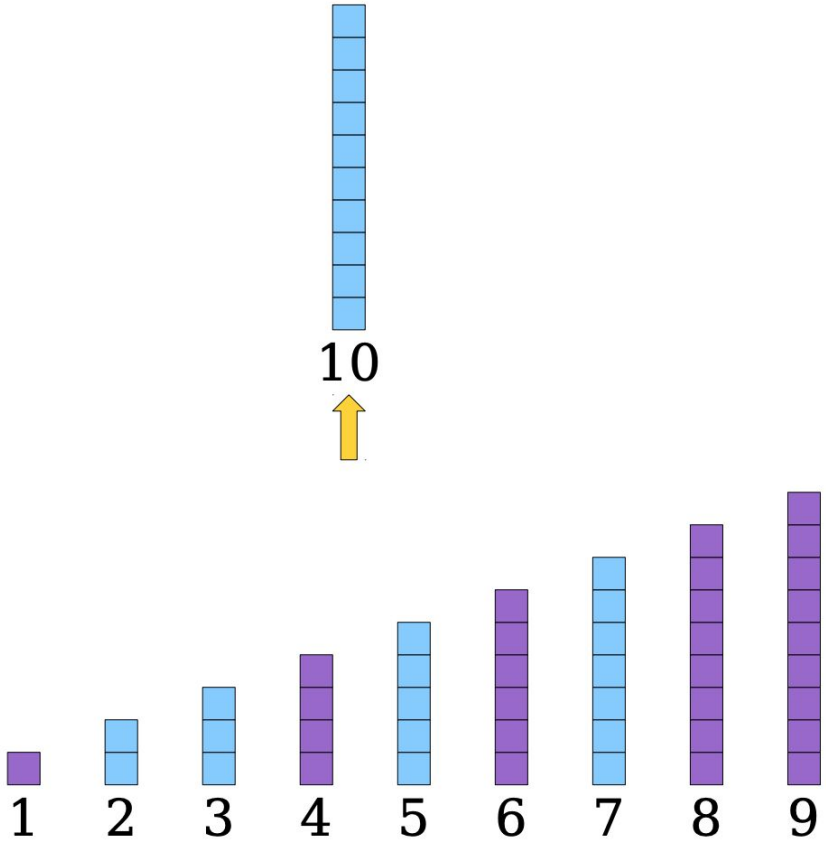
The Key Insight: Merge



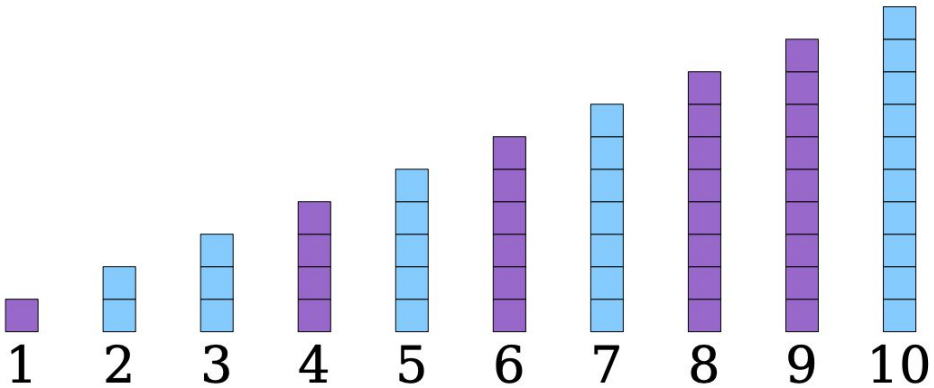
The Key Insight: Merge



The Key Insight: Merge

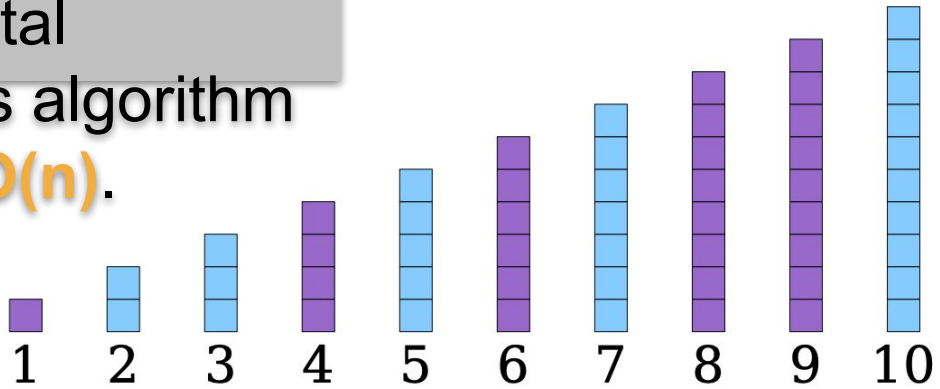


The Key Insight: Merge



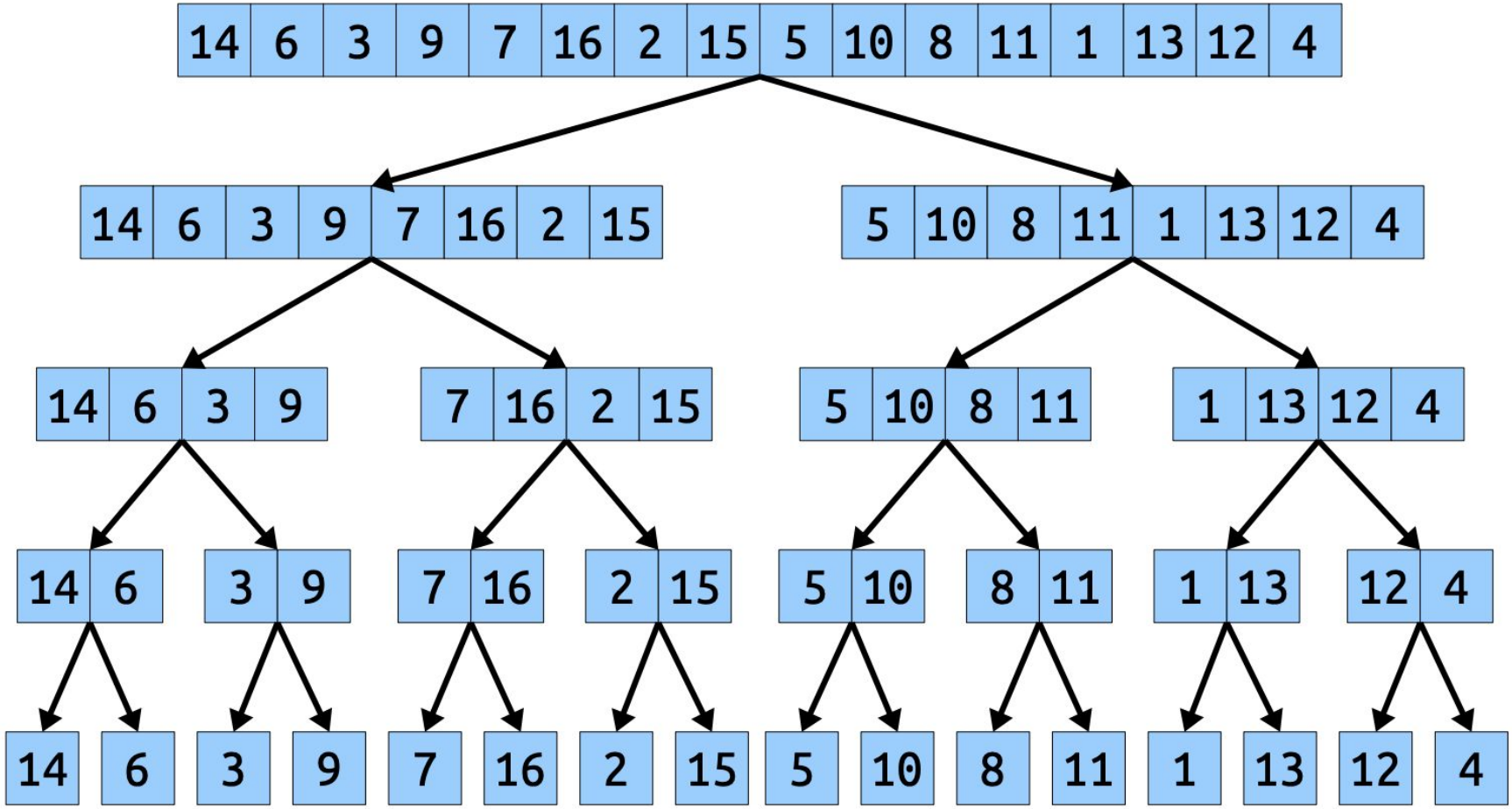
The Key Insight: Merge

Each step makes a single comparison and reduces the number of elements by one. If there are n total elements, this algorithm runs in time **O(n)**.

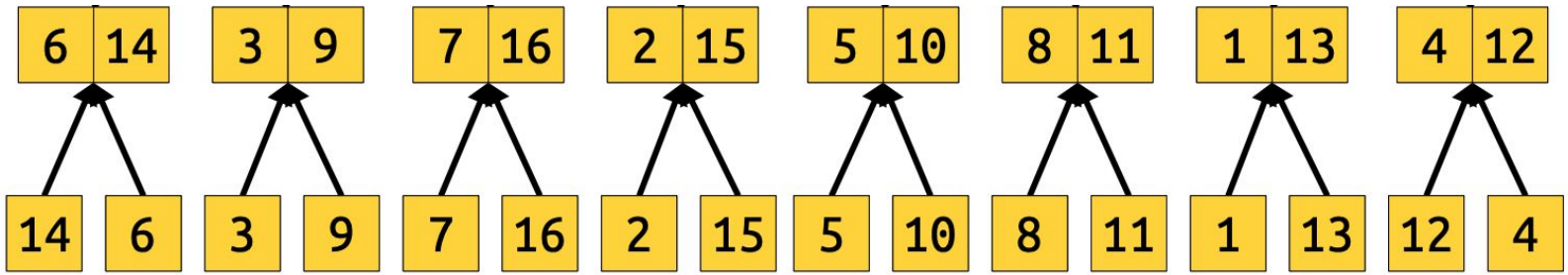


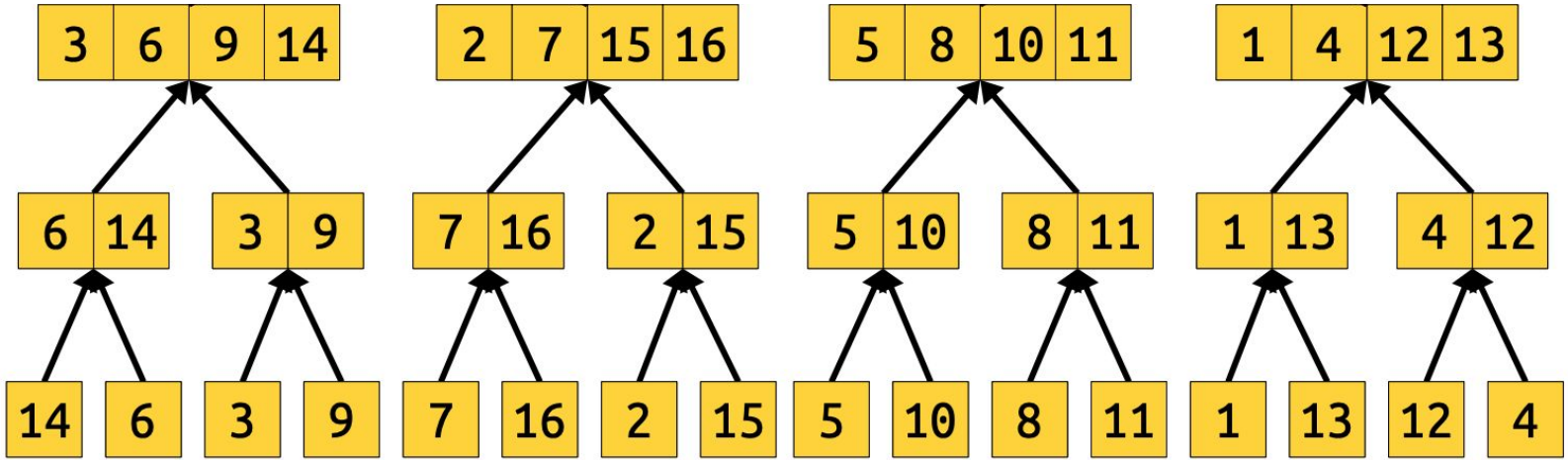
The Key Insight: Merge

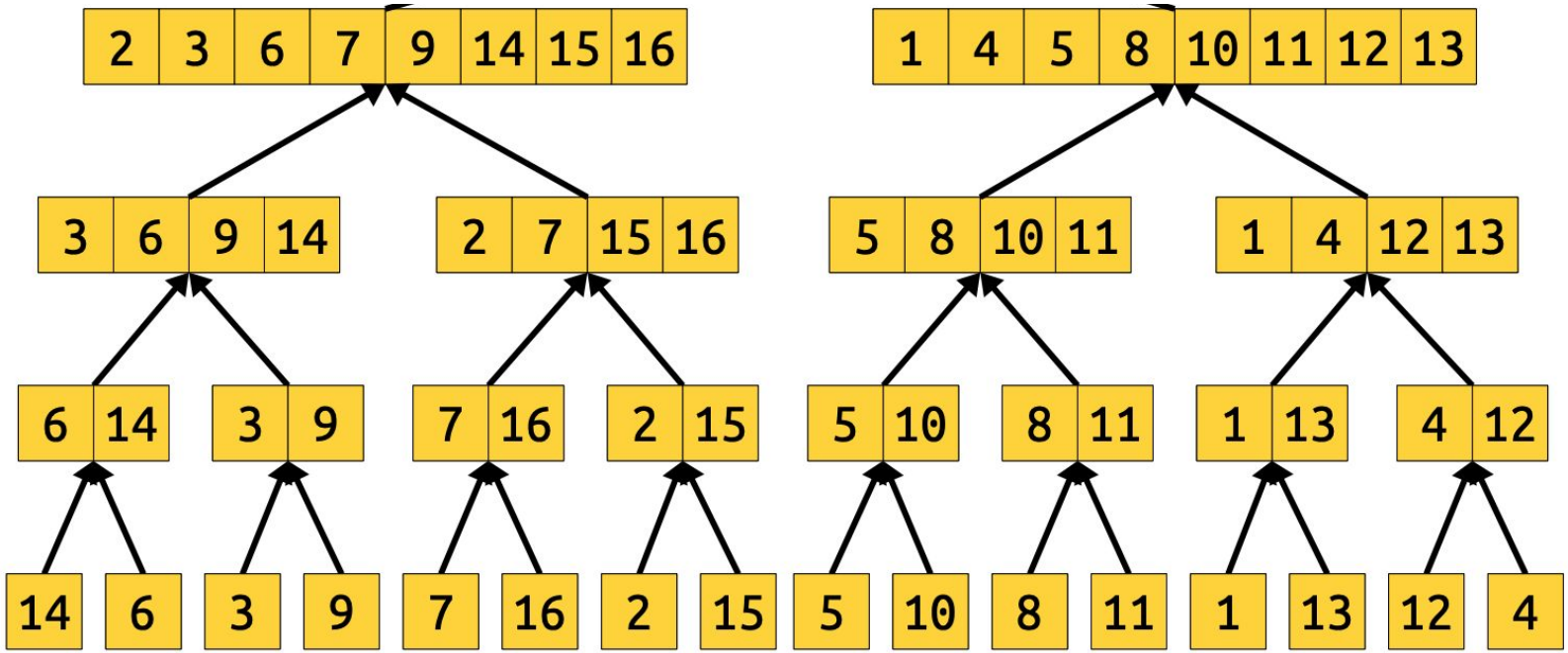
- The merge algorithm takes in two sorted lists and combines them into a single sorted list.
- While both lists are nonempty, compare their first elements. Remove the smaller element and append it to the output.
- Once one list is empty, add all elements from the other list to the output.

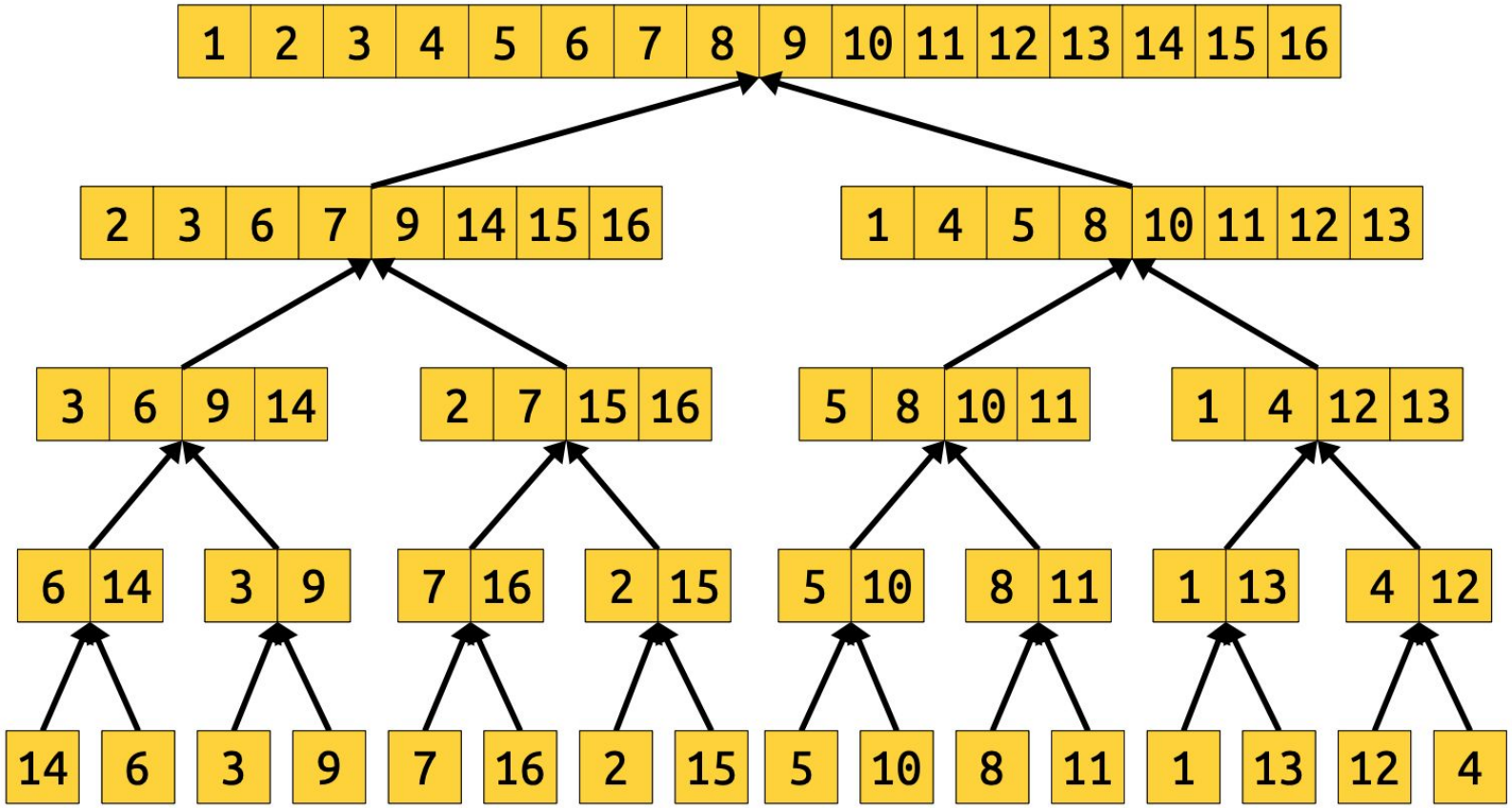

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

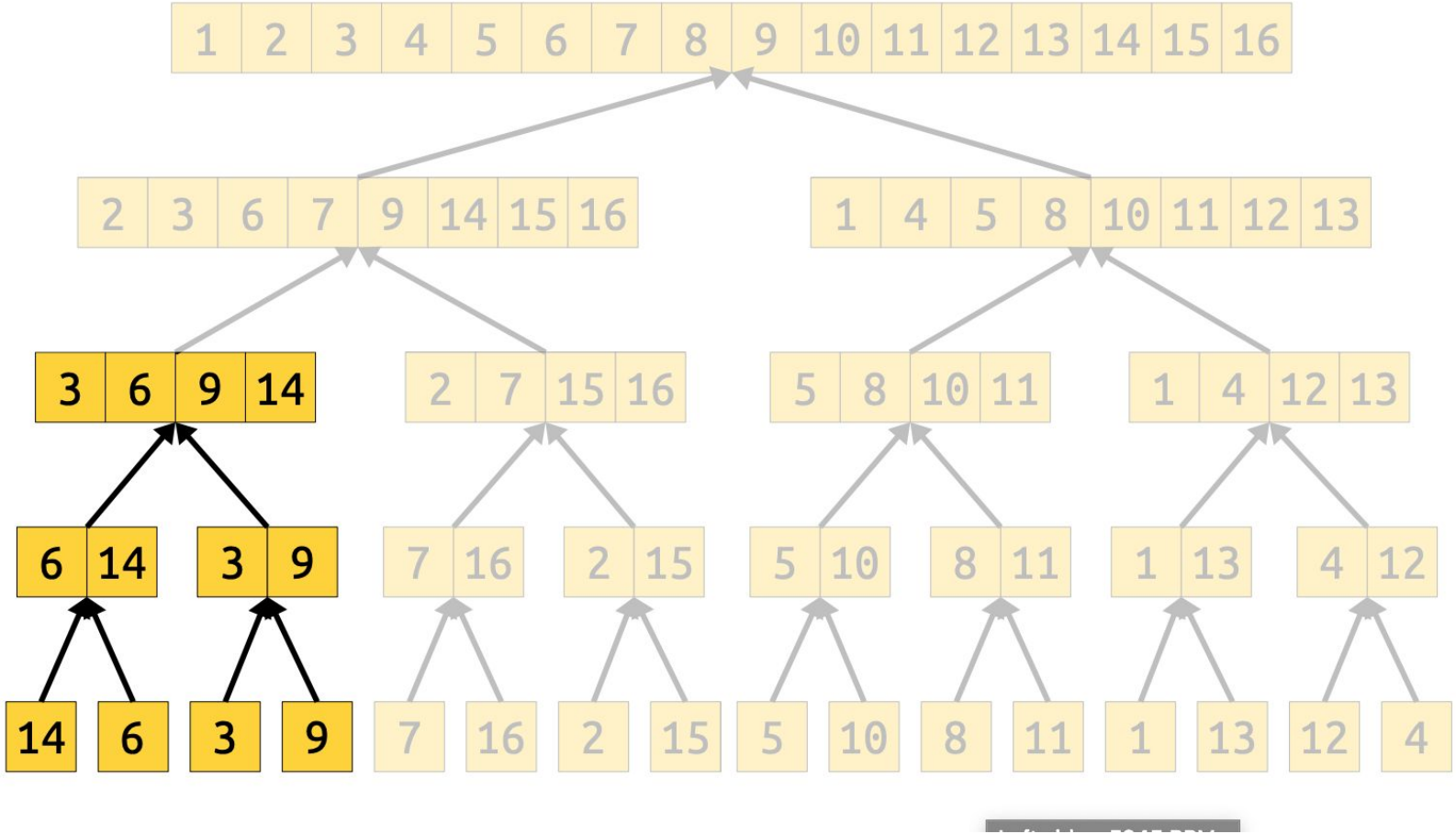

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4


6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4


3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4


2 3 6 7 9 14 15 16
3 6 9 14
2 7 15 16
6 14
3 9
7 16
2 15
14 6 3 9 7 16 2 15
1 4 5 8 10 11 12 13
5 8 10 11
1 4 12 13
5 10
8 11
1 13
4 12
5 10 8 11 1 13 12 4


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
2 3 6 7 9 14 15 16
1 4 5 8 10 11 12 13
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
2 3 6 7 9 14 15 16
1 4 5 8 10 11 12 13
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4

Merge Sort

A recursive sorting algorithm!

- **Base Case:**
  - An empty or single-element list is already sorted.
- **Recursive step:**
  - Break the list in half and recursively sort each part. (easy divide)
  - Use merge to combine them back into a single sorted list (hard join)

Merge Sort – Let's code it!

Analyzing Mergesort:

How fast is this sorting algorithm?

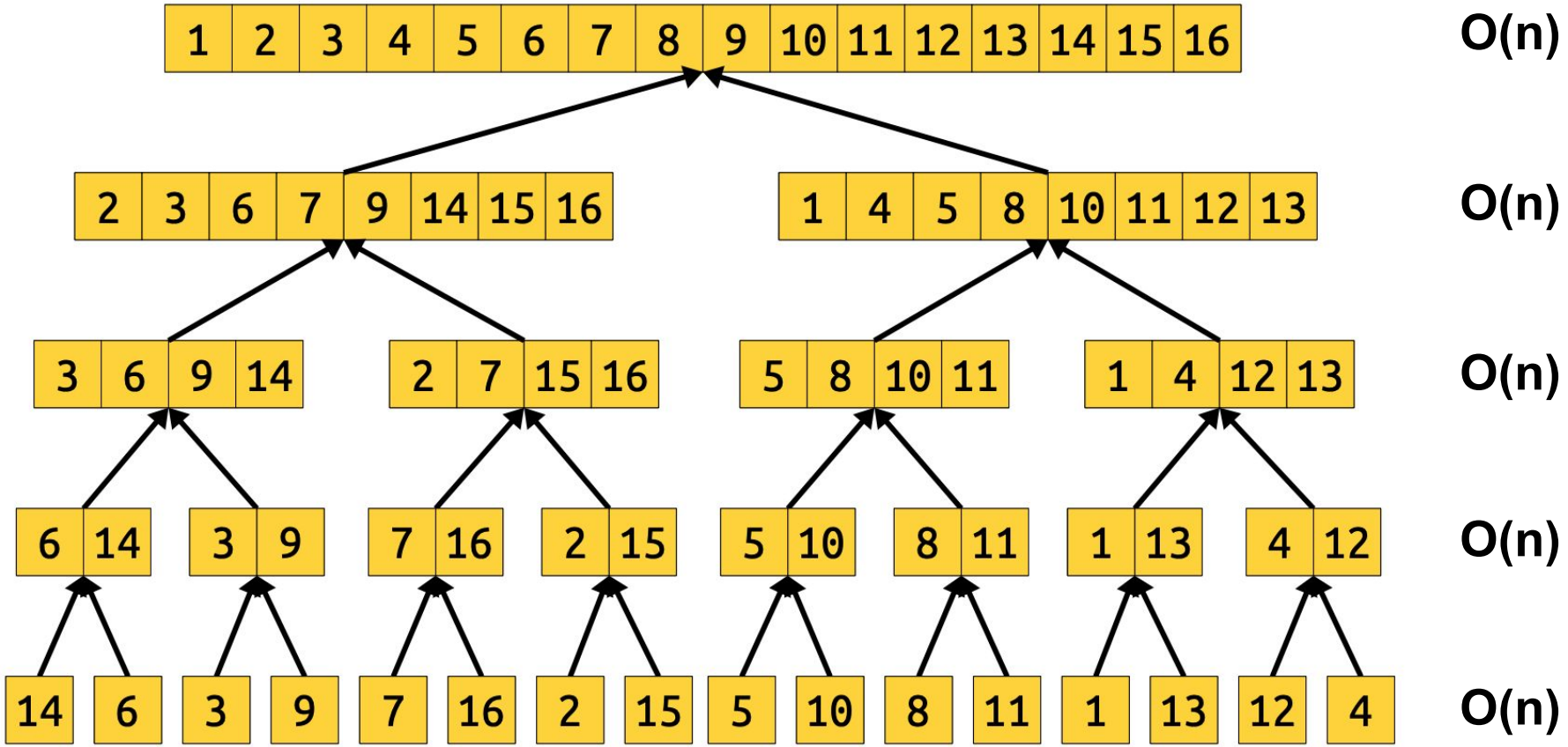
```
void mergeSort(Vector<int>& vec) {
   /* A list with 0 or 1 elements is already sorted by definition. */
   if (vec.size() <= 1) return;

   /* Split the list into two, equally sized halves */
   Vector<int> left, right;
   split(vec, left, right);

   /* Recursively sort the two halves. */
   mergeSort(left);
   mergeSort(right);

   /*
    * Empty out the original vector and re-fill it with merged result
    * of the two sorted halves.
    */
   vec = {};
   merge(vec, left, right);
}
```

```
void mergeSort(Vector<int>& vec) {
   /* A list with 0 or 1 elements is already sorted by definition. */
   if (vec.size() <= 1) return;

   /* Split the list into two, equally sized halves */
   Vector<int> left, right;
   split(vec, left, right);

   /* Recursively sort the two halves. */
   mergeSort(left);
   mergeSort(right);

   /*
    * Empty out the original vector and re-fill it with merged result
    * of the two sorted halves.
    */
   vec = {};
   merge(vec, left, right);
}
```

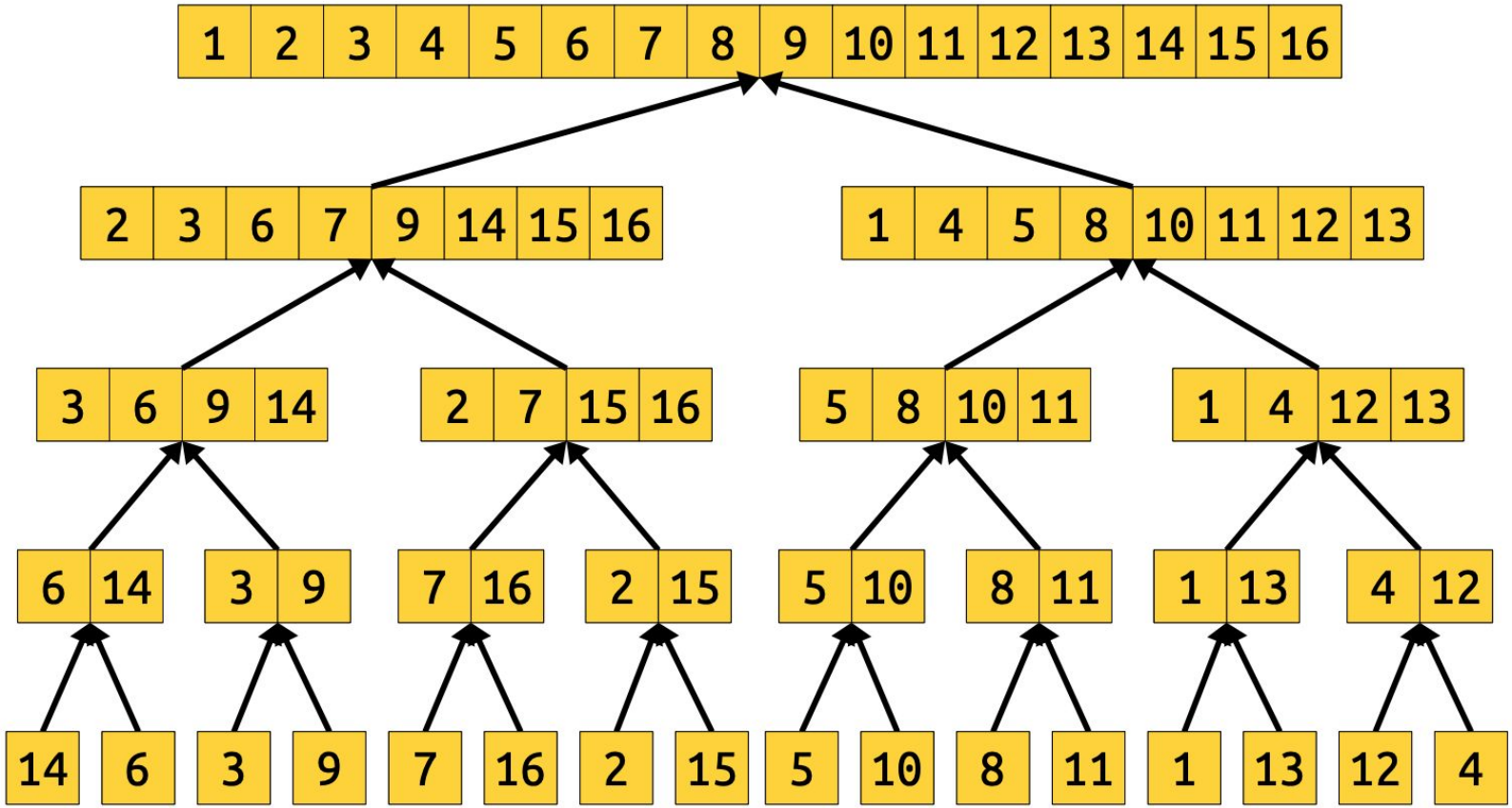
O(n) work

O(n) work

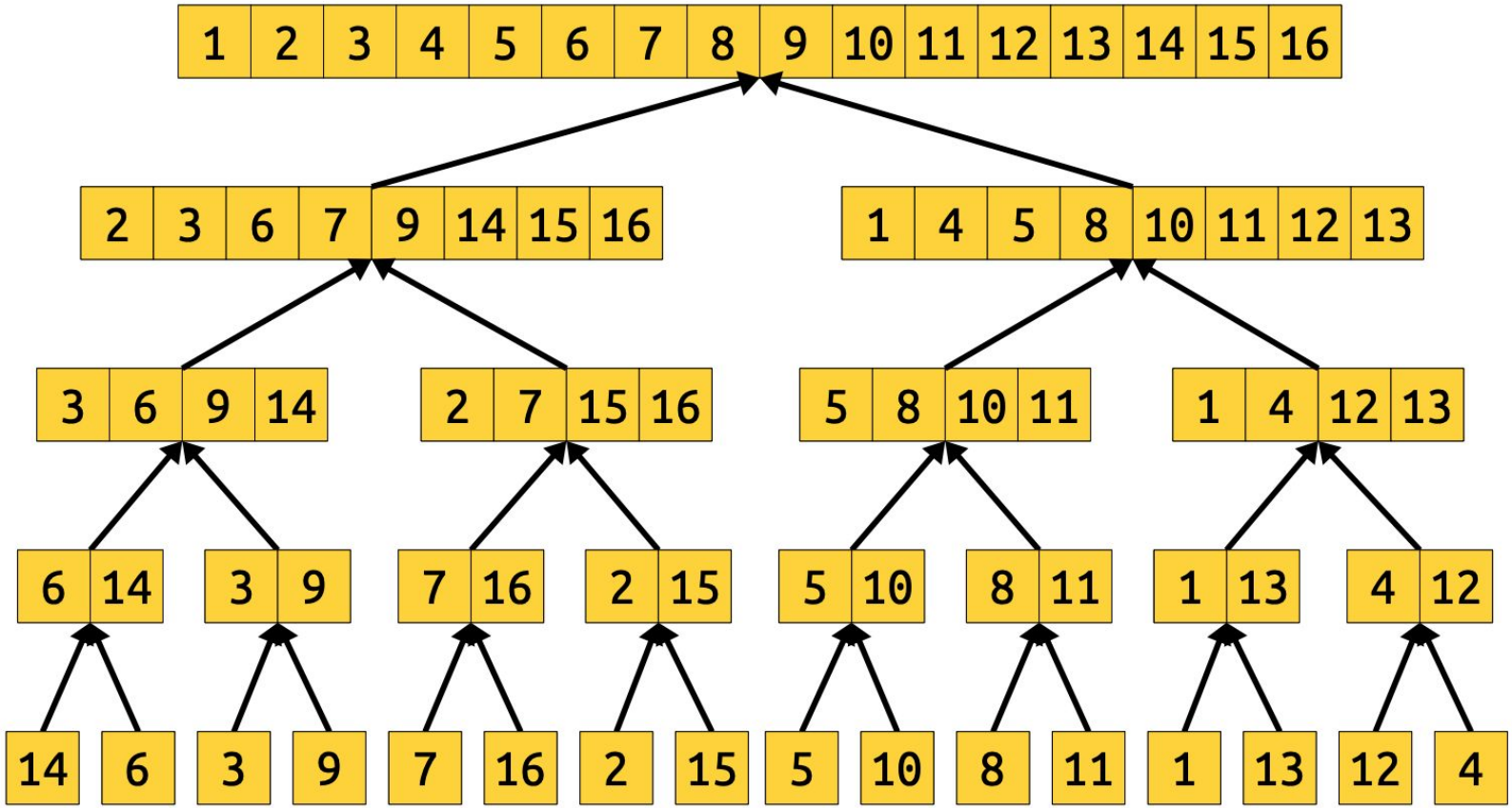
```
void mergeSort(Vector<int>& vec) {
   /* A list with 0 or 1 elements is already sorted by definition. */
   if (vec.size() <= 1) return;

   /* Split the list into two, equally sized halves */
   Vector<int> left, right;
   split(vec, left, right);

   /* Recursively sort the two halves. */
   mergeSort(left);
   mergeSort(right);

   /*
    * Empty out the original vector and re-fill it with merged result
    * of the two sorted halves.
    */
   vec = {};
   merge(vec, left, right);
}
```



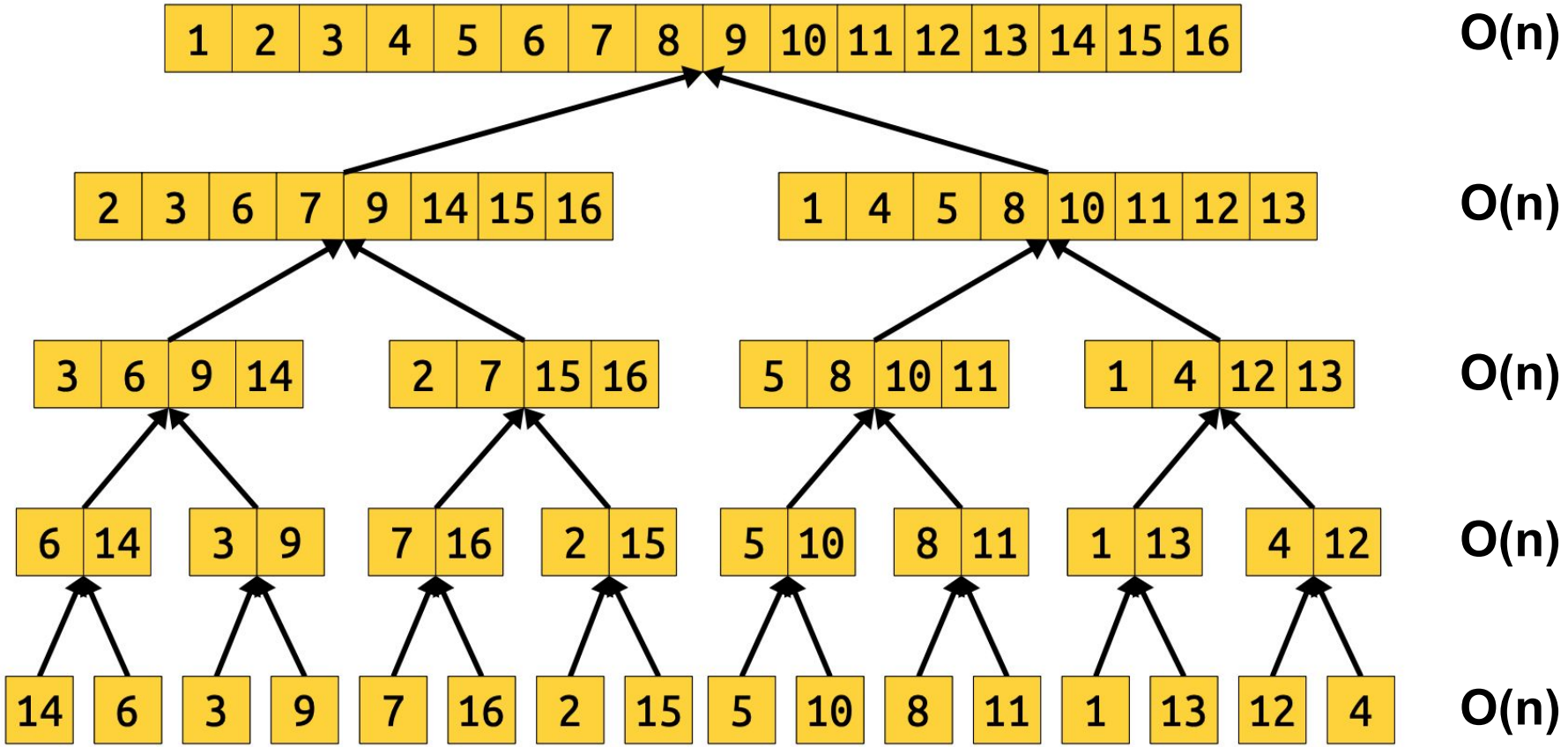
O(n) work at each level!



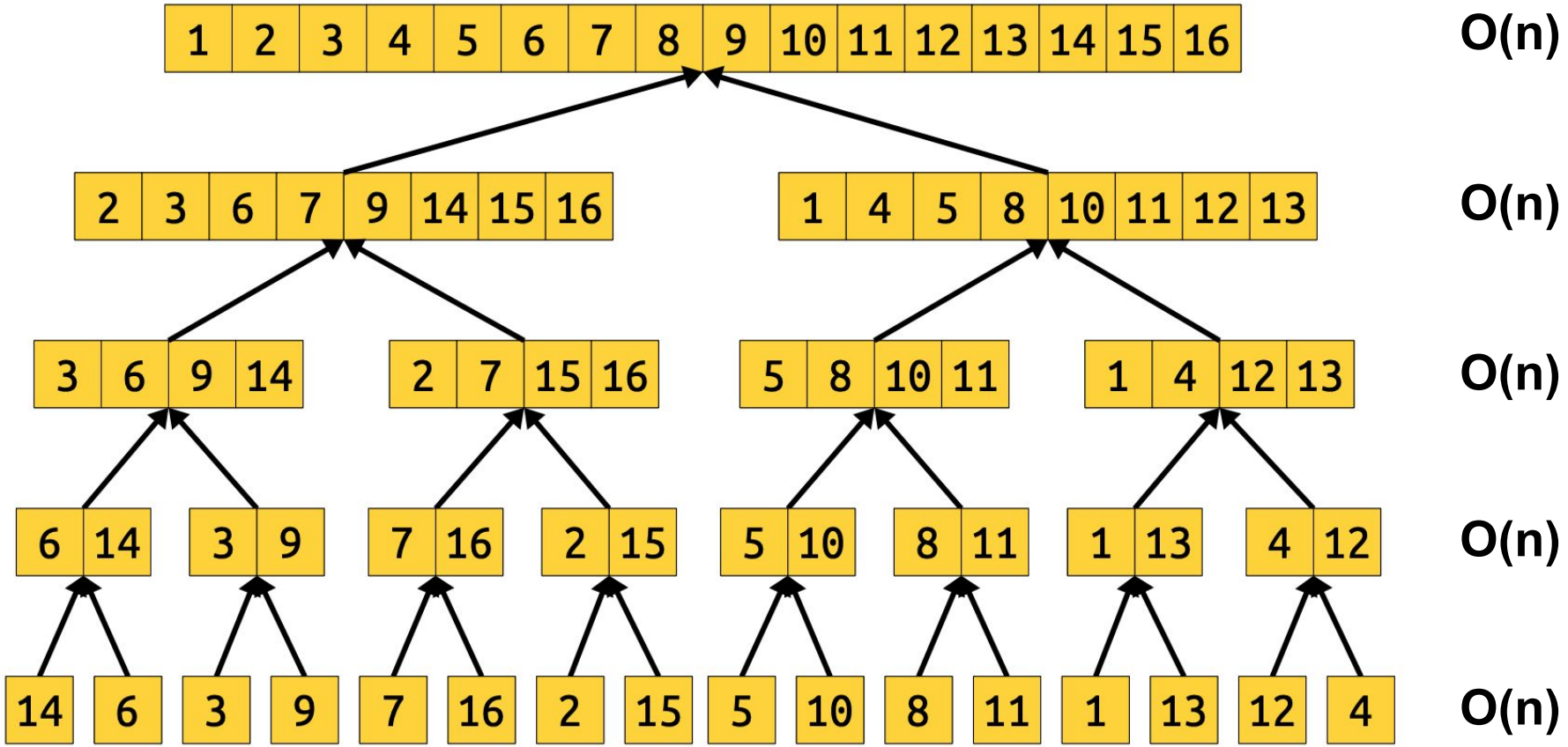
How many levels are there?



Remember: How many times do we divide by 2?



O(log n) levels!



Total work: **O(n * log n)**

```
void mergeSort(Vector<int>& vec) {
   /* A list with 0 or 1 elements is already sorted by definition. */
   if (vec.size() <= 1) return;

   /* Split the list into two, equally sized halves */
   Vector<int> left, right;
   split(vec, left, right);

   /* Recursively sort the two halves. */
   mergeSort(left);
   mergeSort(right);

   /*
    * Empty out the original vector and re-fill it with merged result
    * of the two sorted halves.
    */
   vec = {};
   merge(vec, left, right);
}
```

O(n log n) work

Analyzing Mergesort: Can we do better?

- Mergesort runs in time O(n log n), which is faster than insertion sort's $O(n^2)$.
  - Can we do better than this?

- A comparison sort is a sorting algorithm that only learns the relative ordering of its elements by making comparisons between elements.
  - All of the sorting algorithms we've seen so far are comparison sorts.

- **Theorem**: There are no comparison sorts whose average-case runtime can be better than O(n log n).

- If we stick with making comparisons, we can only hope to improve on mergesort by a constant factor!

A Quick Historical Aside

- Mergesort was one of the first algorithms developed for computers as we know them today.

- It was invented by John von Neumann in 1945 (!) as a way of validating the design of the first “modern” (stored-program) computer.

- Want to learn more about what he did? Check out this article by Stanford’s very own Donald Knuth.

Announcements

Announcements

- Assignment 5 was released yesterday and will be due on **Tuesday, August 3 at 11:59pm PDT**.

- The Assignment 5 YEAH session will take place **Friday 7/30** at **11:30am PDT.**

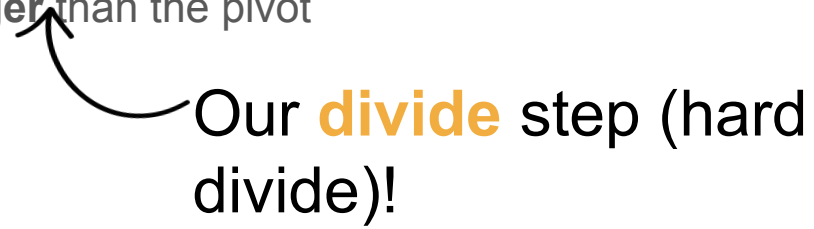
Quicksort

Quicksort Algorithm

1. **Partition** the elements into three categories based on a chosen **pivot** element:
    - Elements **smaller** than the pivot
    - Elements **equal** to the pivot
    - Elements **larger** than the pivot

Quicksort Algorithm

1. **Partition** the elements into three categories based on a chosen **pivot** element:
    - Elements **smaller** than the pivot
    - Elements **equal** to the pivot
    - Elements **larger** than the pivot



Quicksort Algorithm

1. **Partition** the elements into three categories based on a chosen **pivot** element:
    - Elements **smaller** than the pivot
    - Elements **equal** to the pivot
    - Elements **larger** than the pivot

2. **Recursively sort** the two partitions that are not equal to the pivot (smaller and larger elements).
    - Now our smaller elements are in sorted order, and our larger elements are also in sorted order!

Quicksort Algorithm

1. **Partition** the elements into three categories based on a chosen **pivot** element:
    - Elements **smaller** than the pivot
    - Elements **equal** to the pivot
    - Elements **larger** than the pivot

2. **Recursively sort** the two partitions that are not equal to the pivot (smaller and larger elements).
    - Now our smaller elements are in sorted order, and our larger elements are also in sorted order!

3. **Concatenate** the three now-sorted partitions together.

Quicksort Algorithm

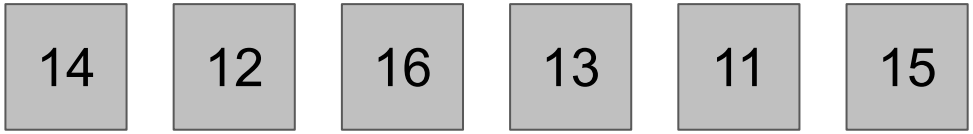
1. **Partition** the elements into three categories based on a chosen **pivot** element:
    - Elements **smaller** than the pivot
    - Elements **equal** to the pivot
    - Elements **larger** than the pivot

2. **Recursively sort** the two partitions that are not equal to the pivot (smaller and larger elements).
    - Now our smaller elements are in sorted order, and our larger elements are also in sorted order!

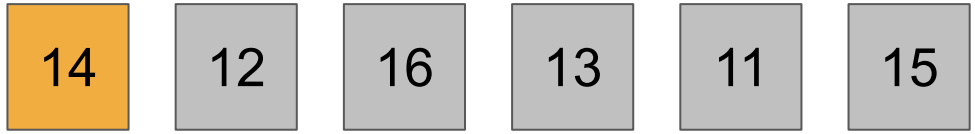
3. **Concatenate** the three now-sorted partitions together.

Our **join** step!
(easy join)

Input of unsorted elements:

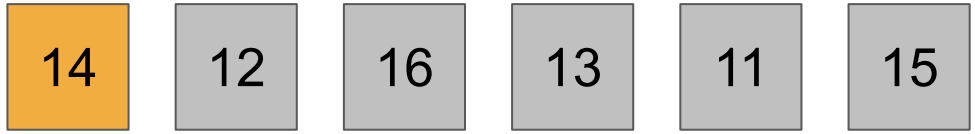


Input of unsorted elements:

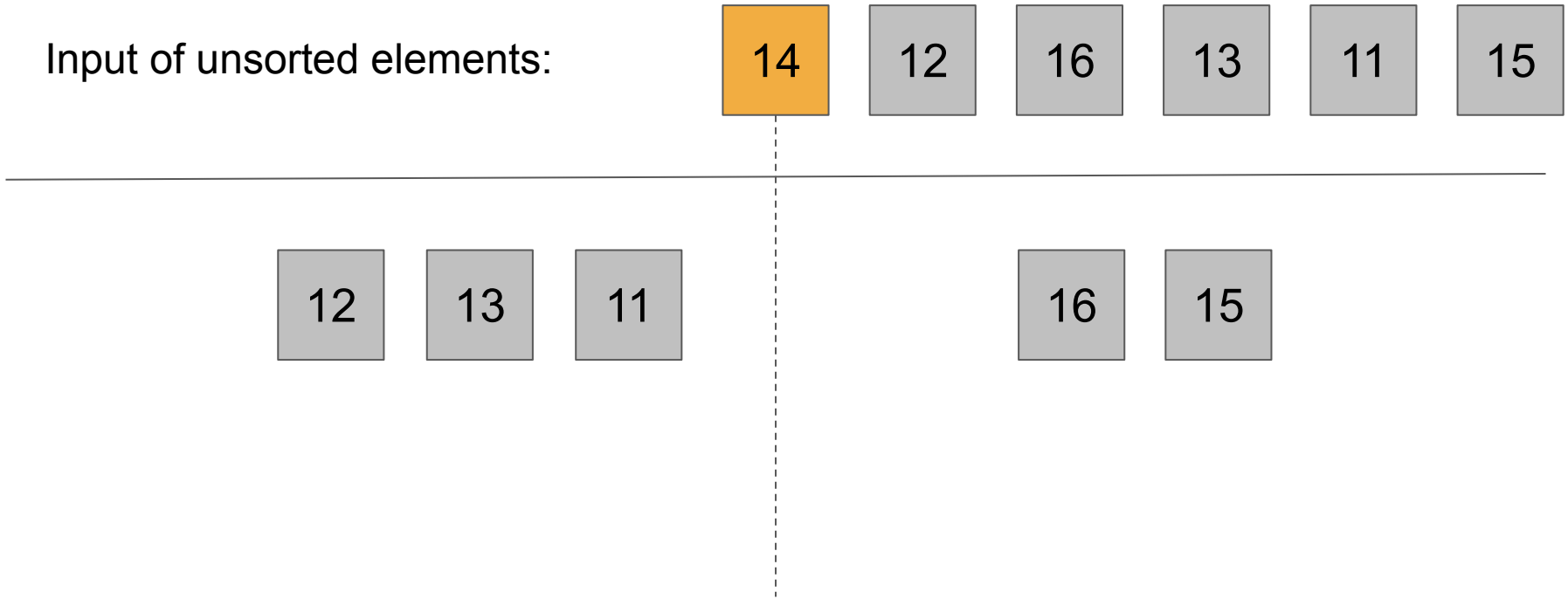


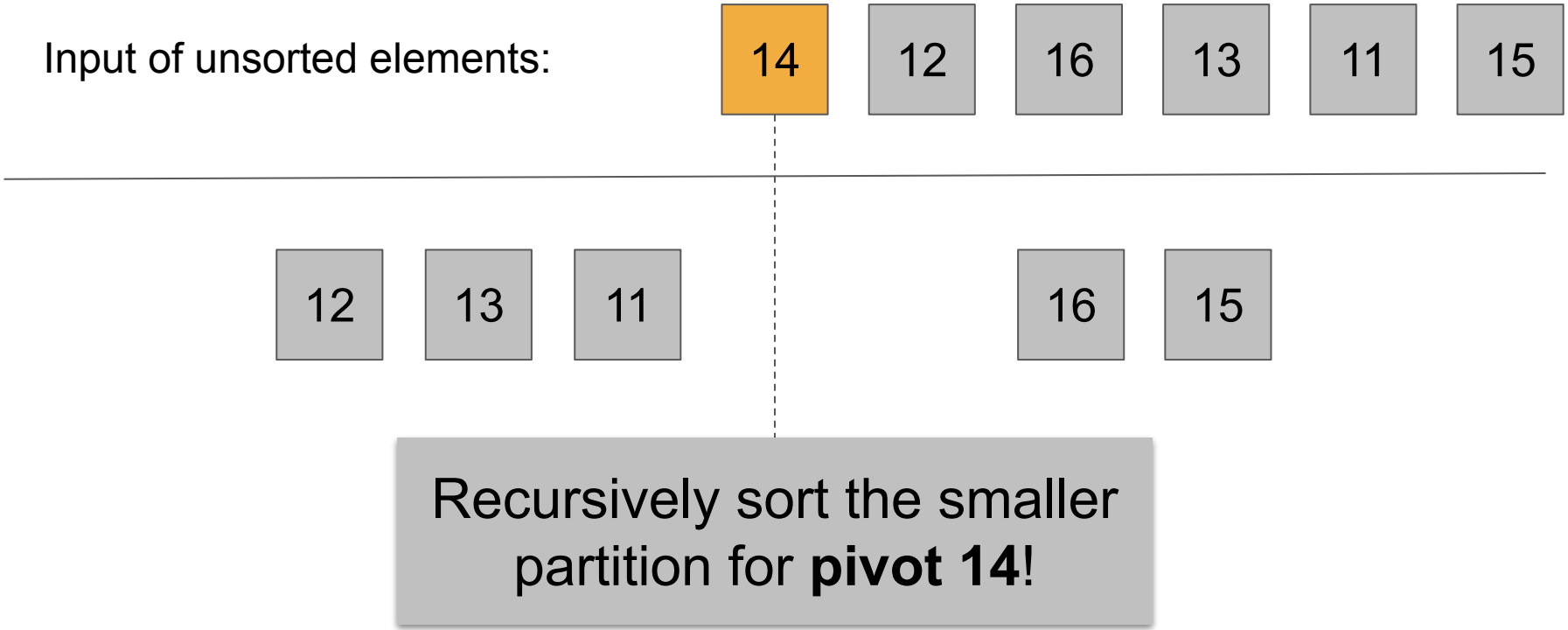
Choose the first element as the pivot.

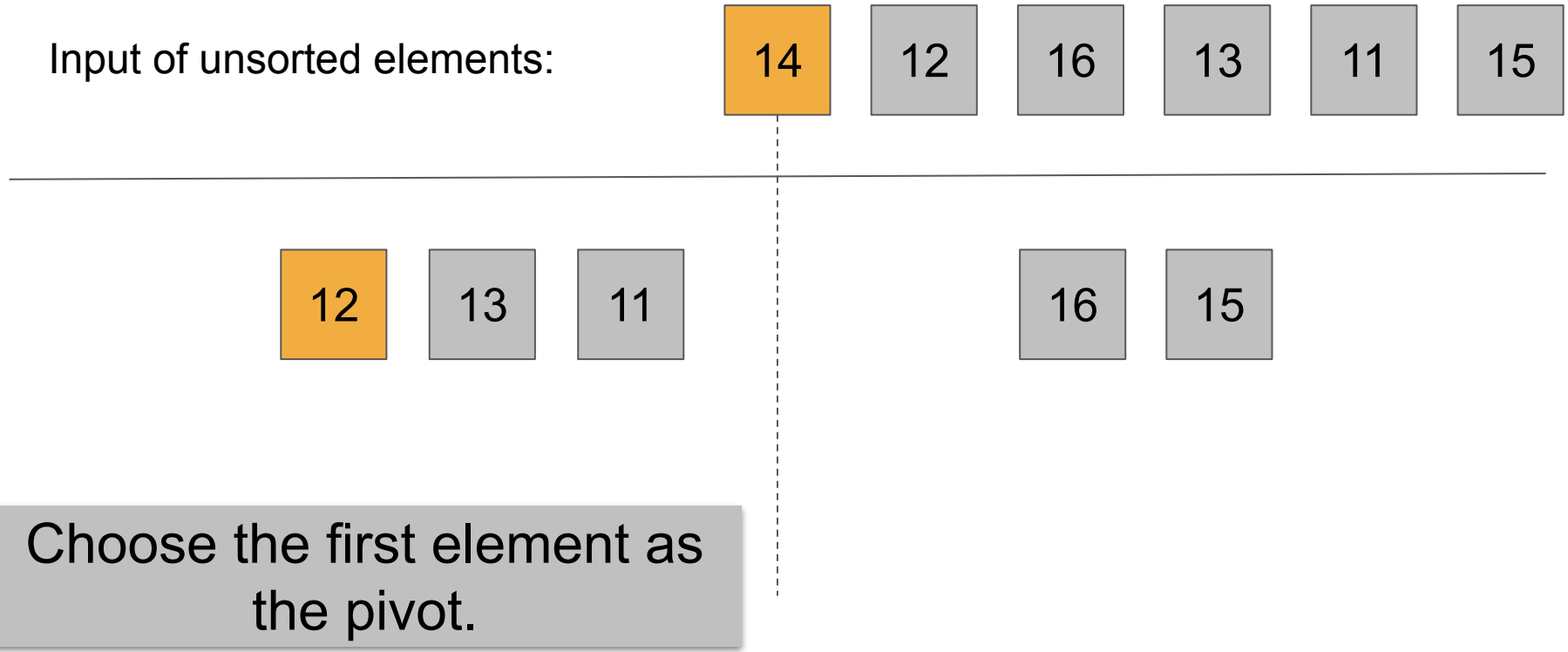
Input of unsorted elements:

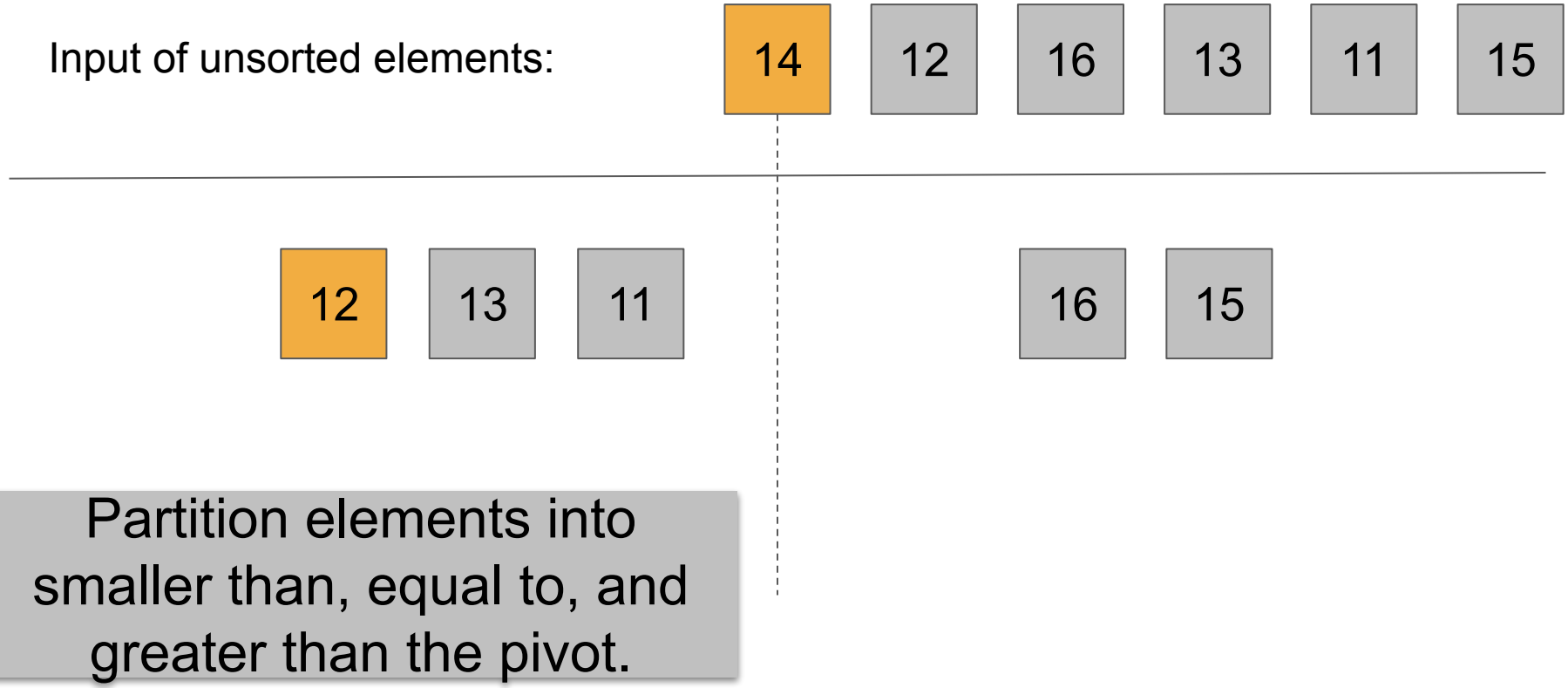


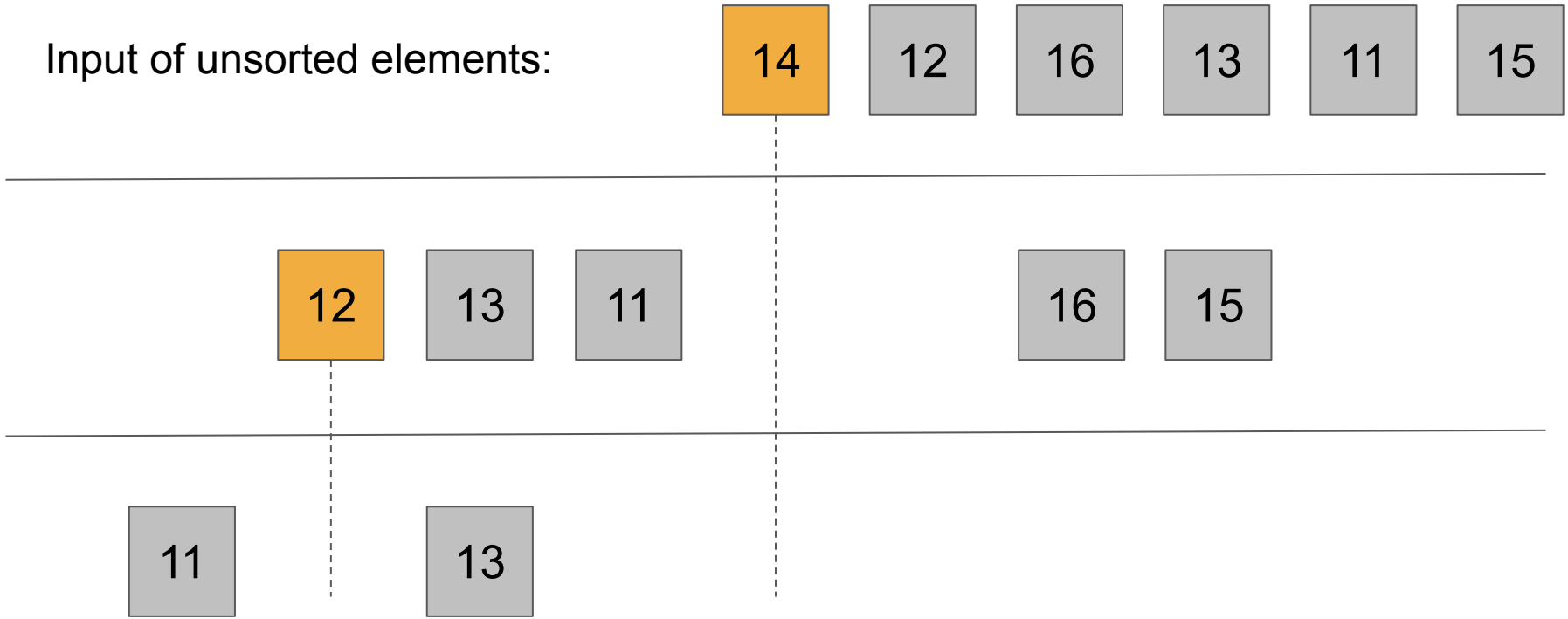
Partition elements into smaller than, equal to, and greater than the pivot.

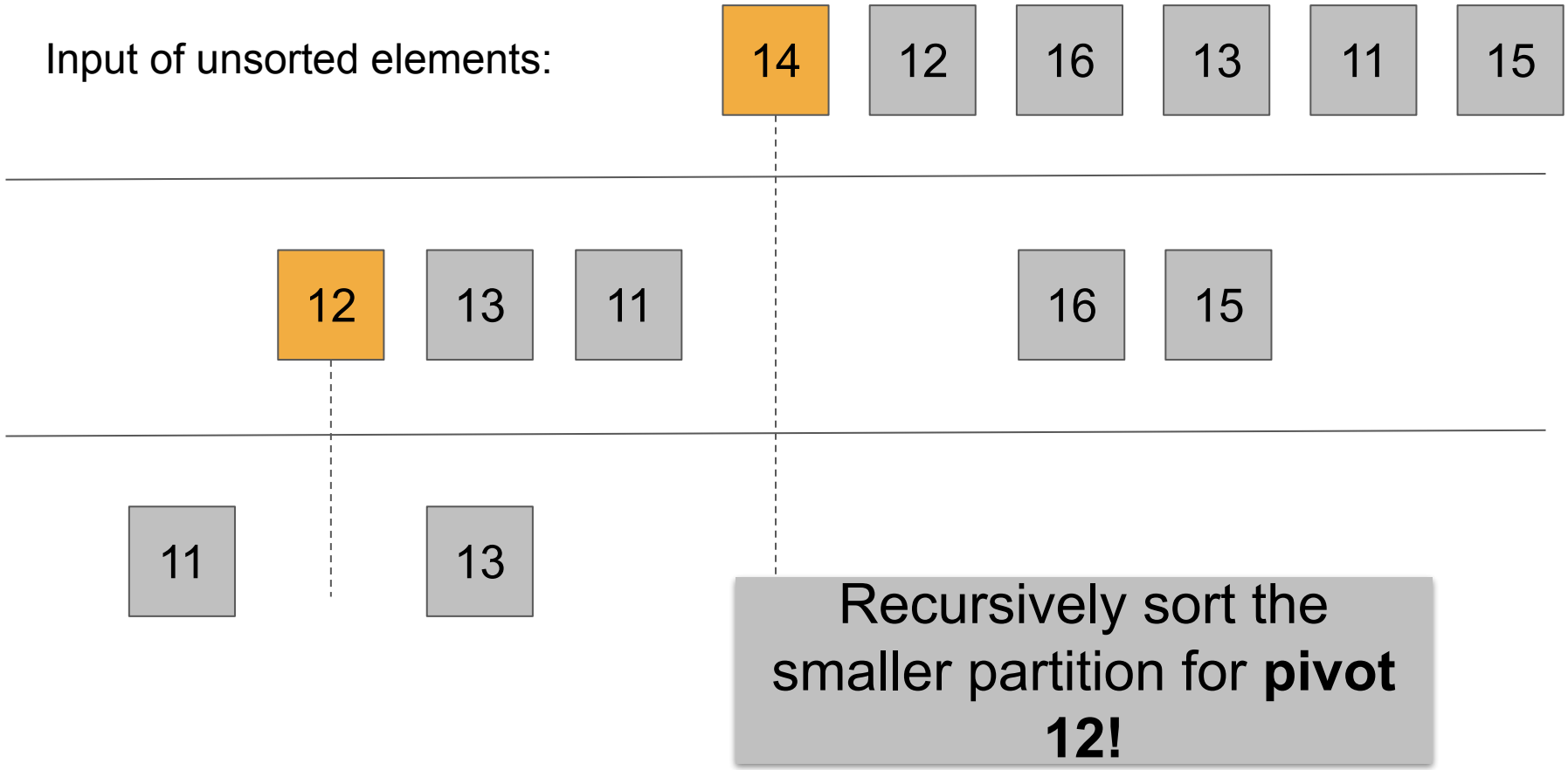

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15

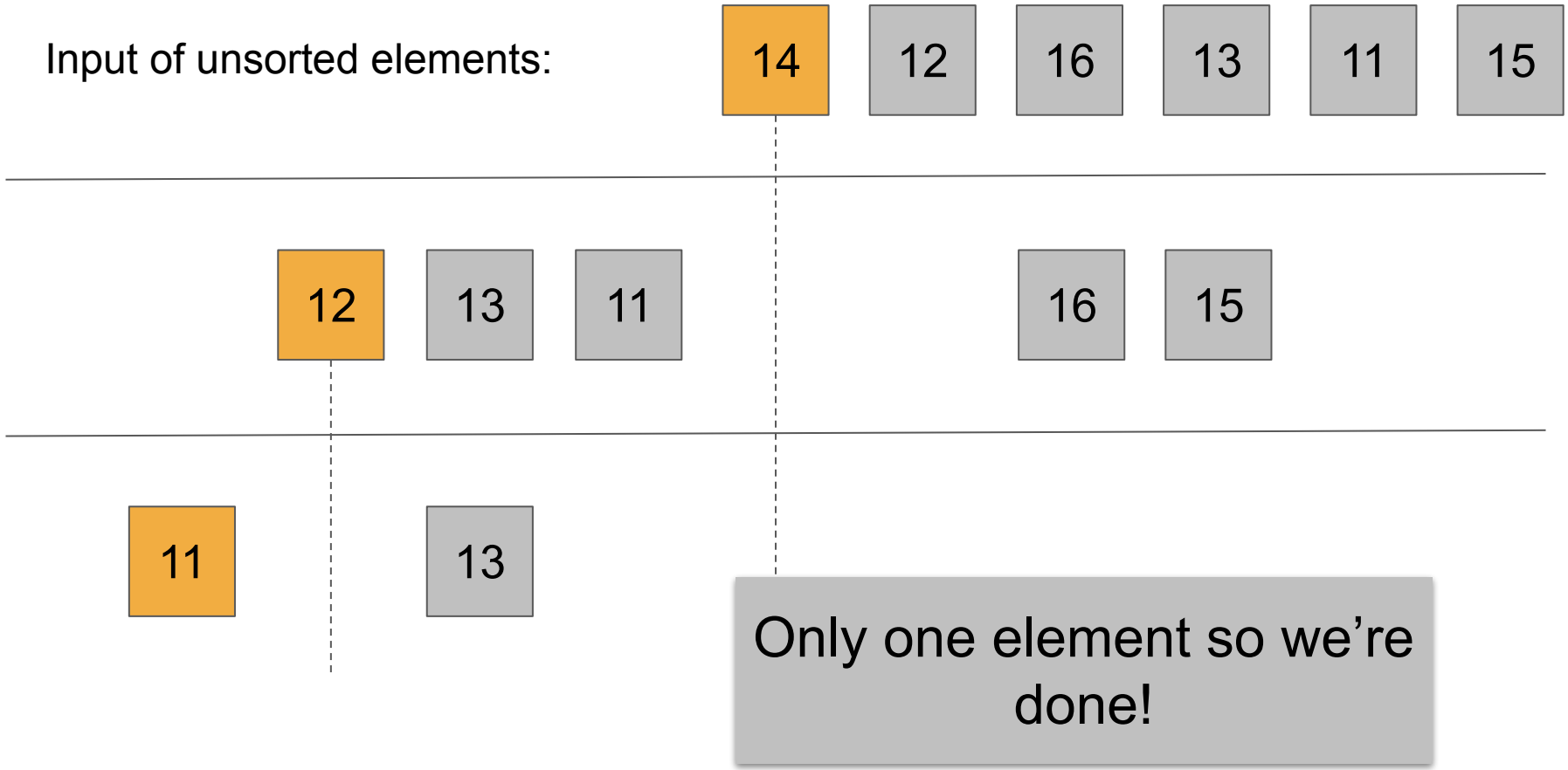

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
Recursively sort the smaller partition for **pivot 14**!

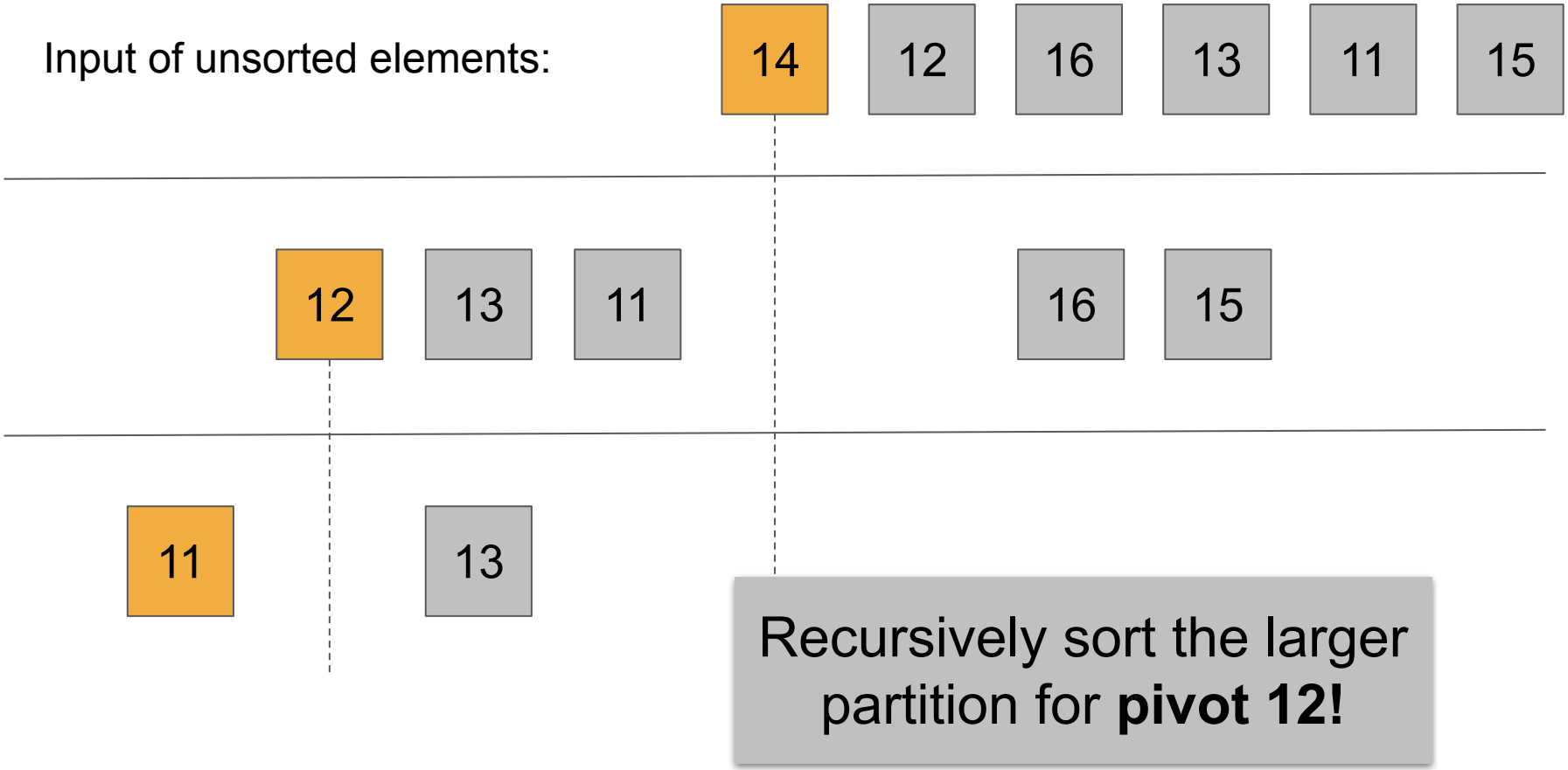

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
Choose the first element as the pivot.

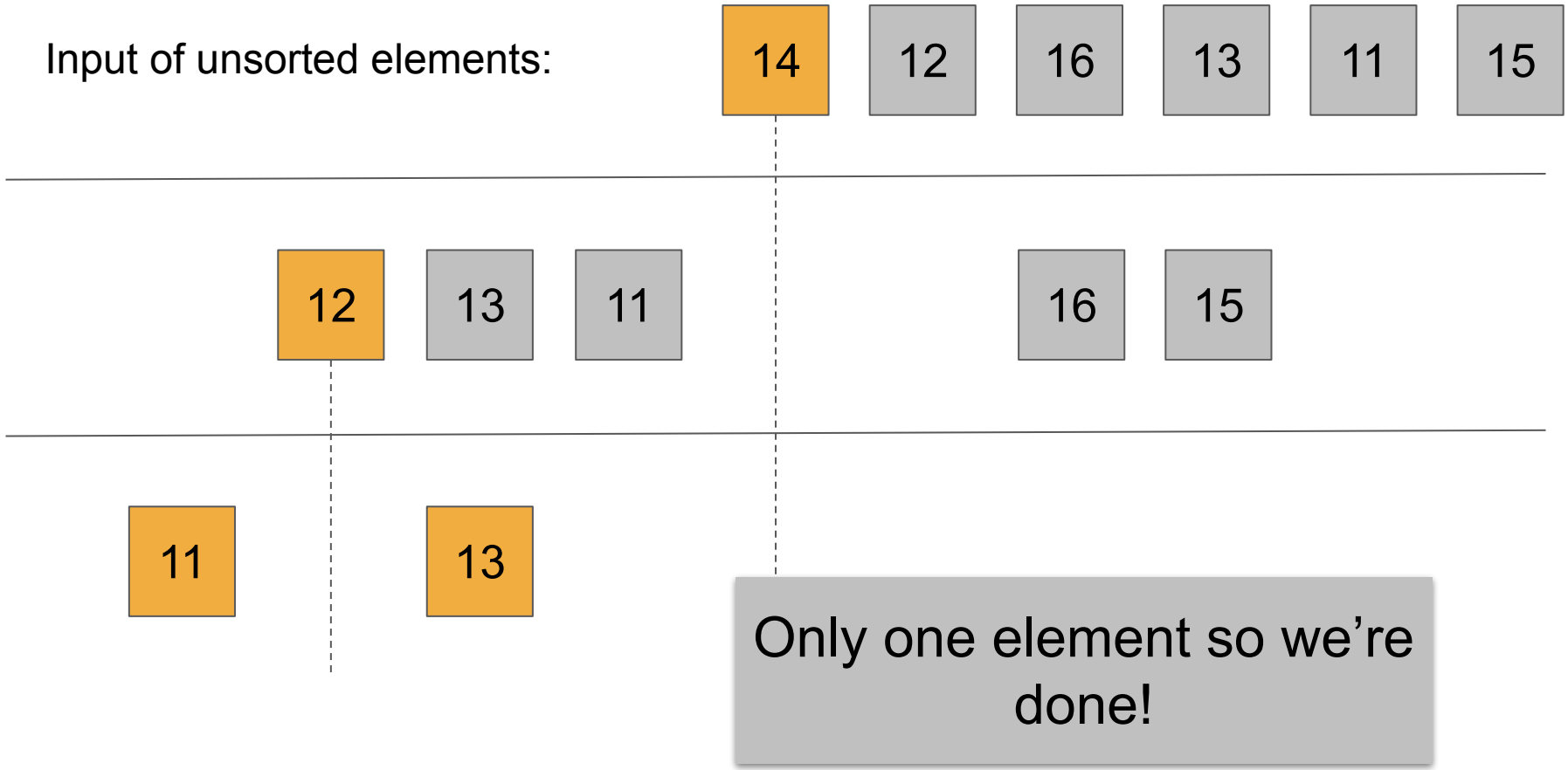

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
Partition elements into smaller than, equal to, and greater than the pivot.

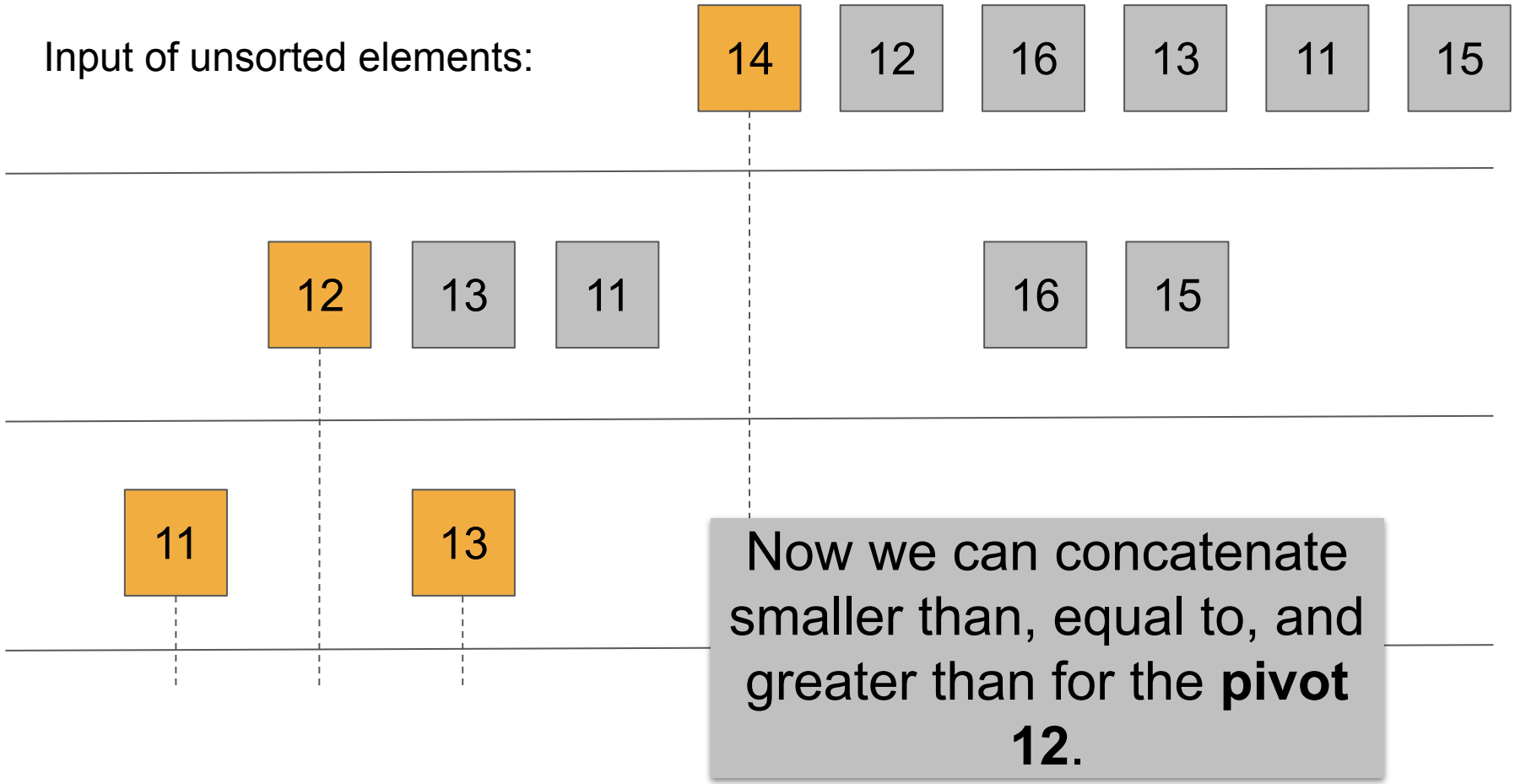

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13

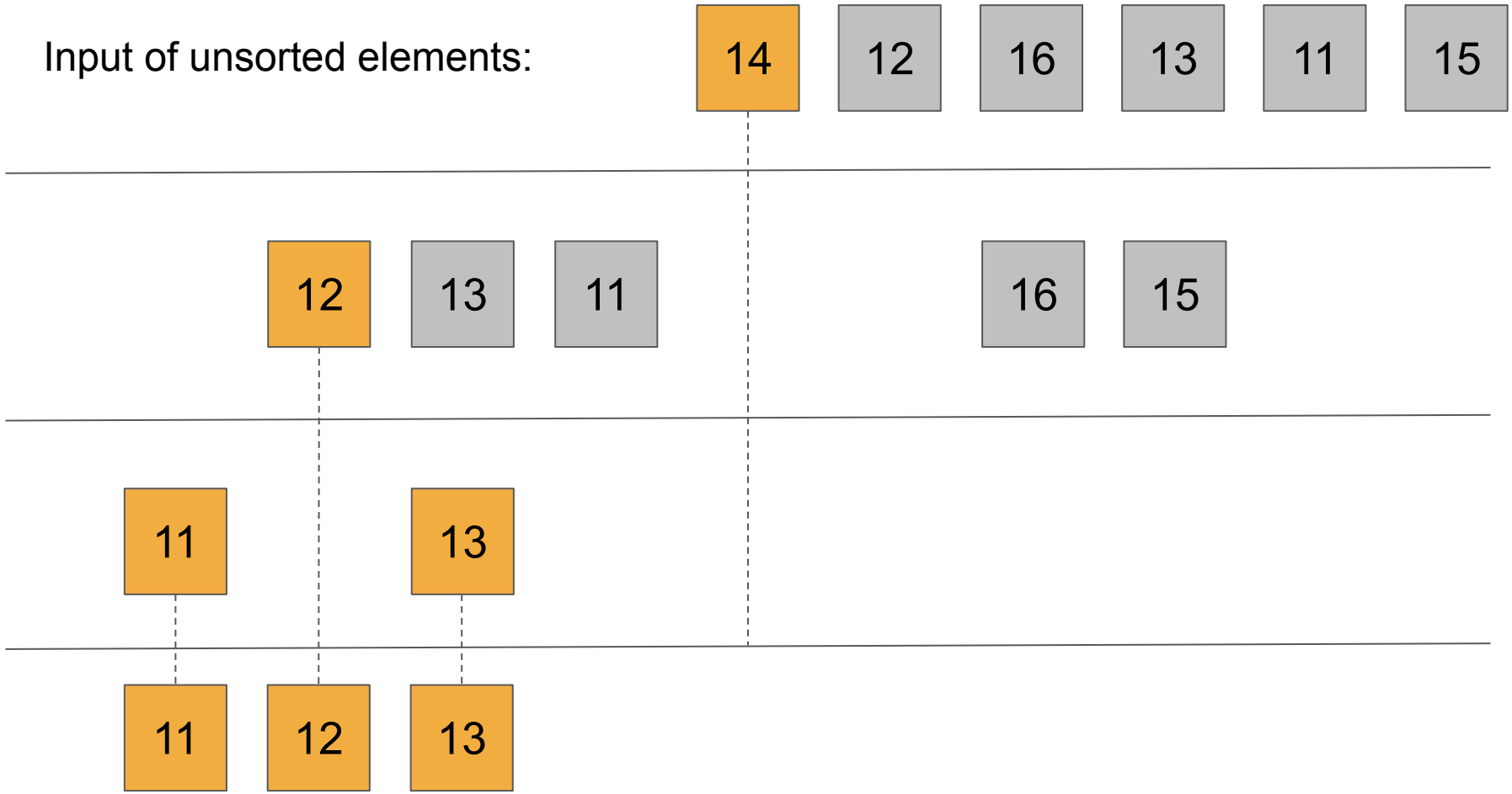

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
Recursively sort the smaller partition for **pivot 12!**

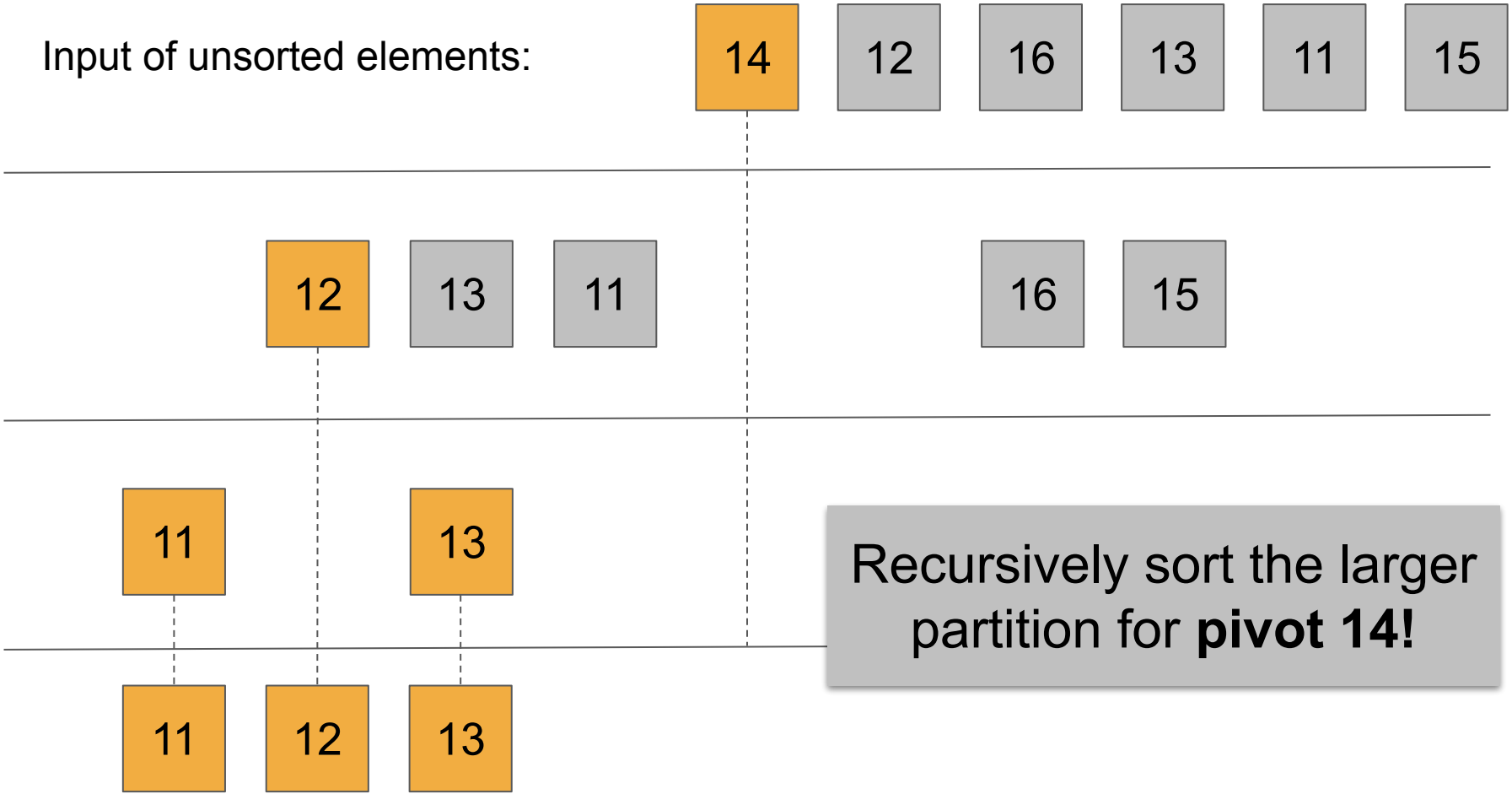

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
Only one element so we're done!

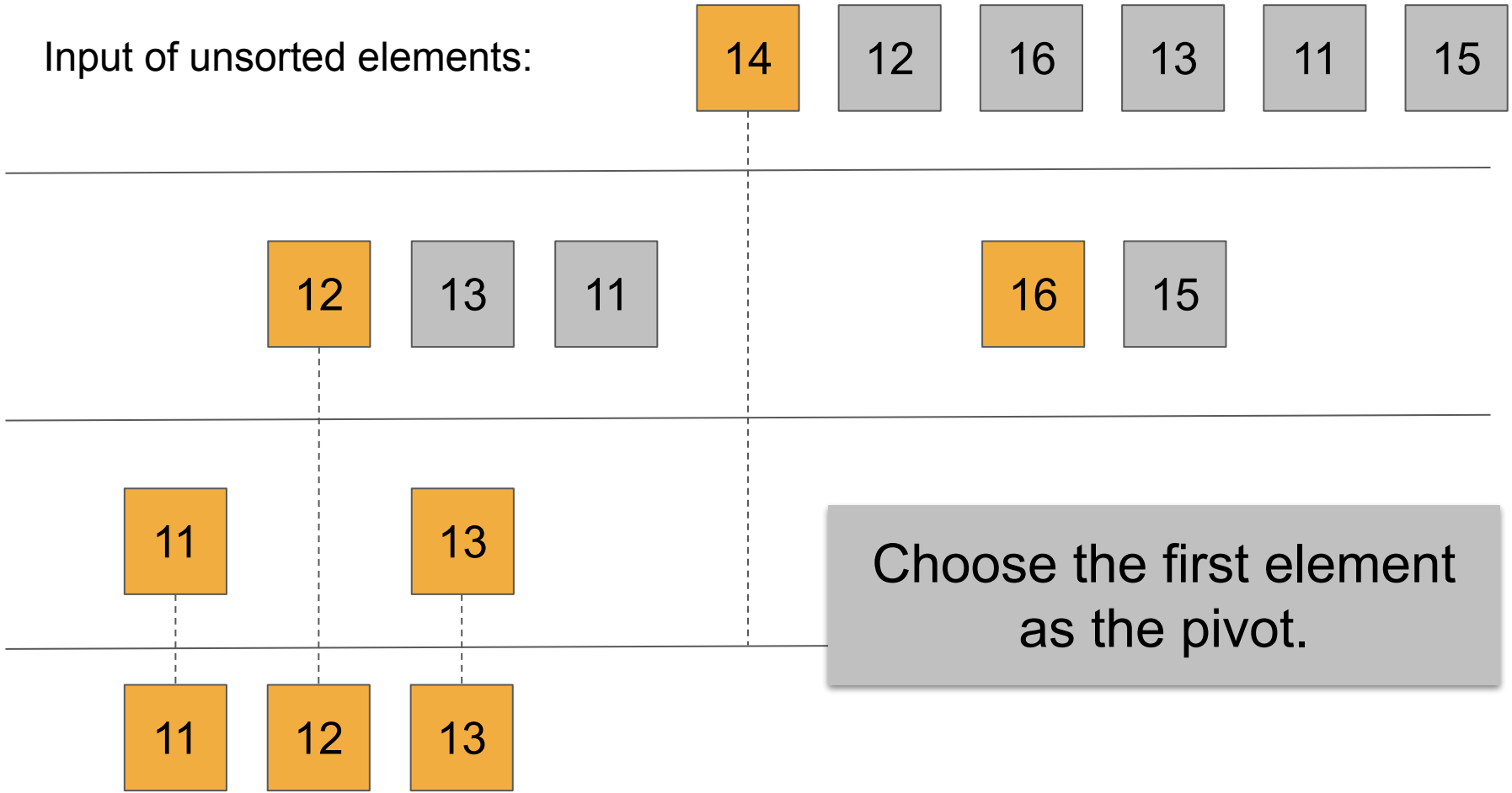

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
Recursively sort the larger partition for **pivot 12!**

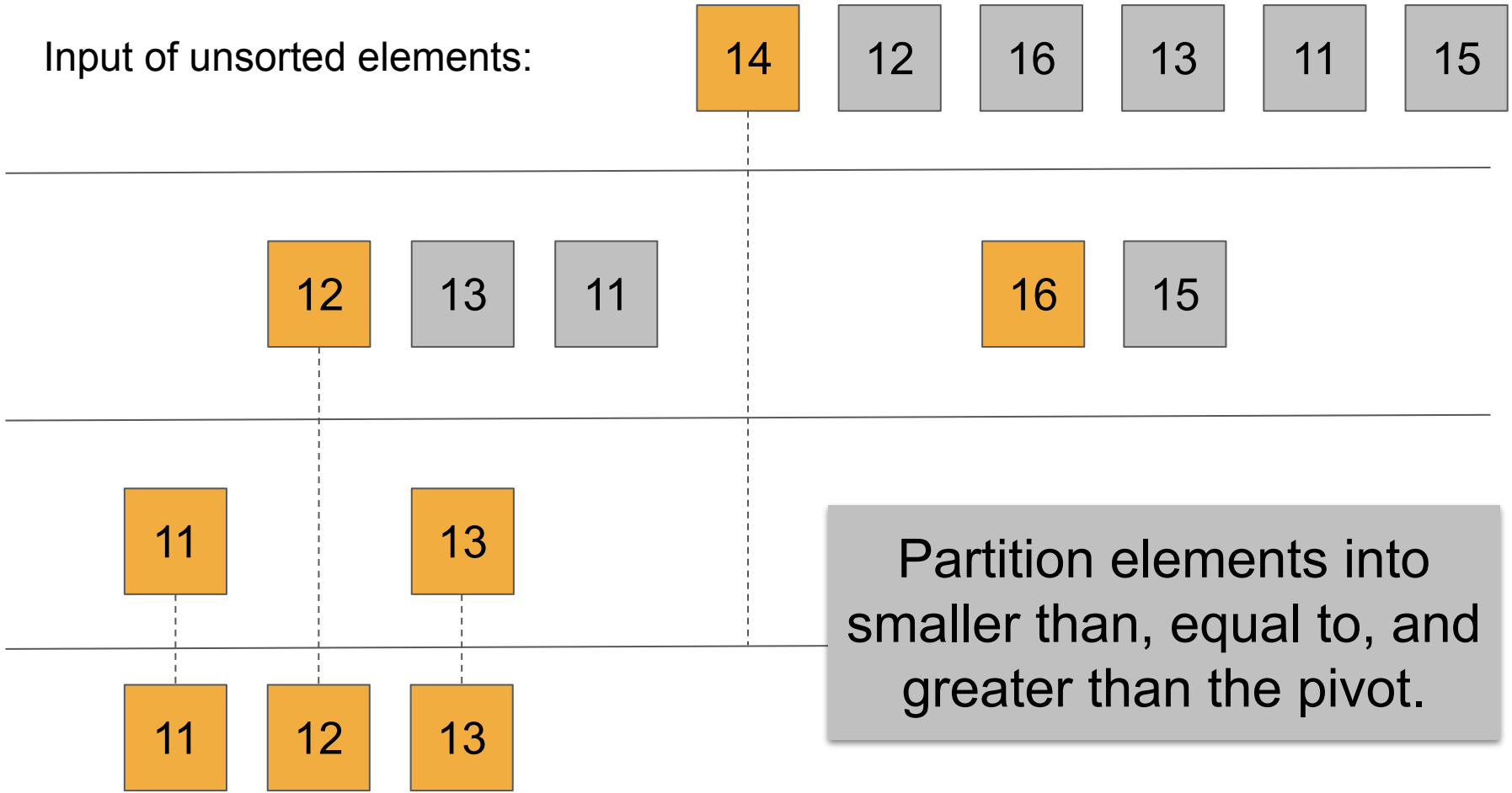

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
Only one element so we're done!

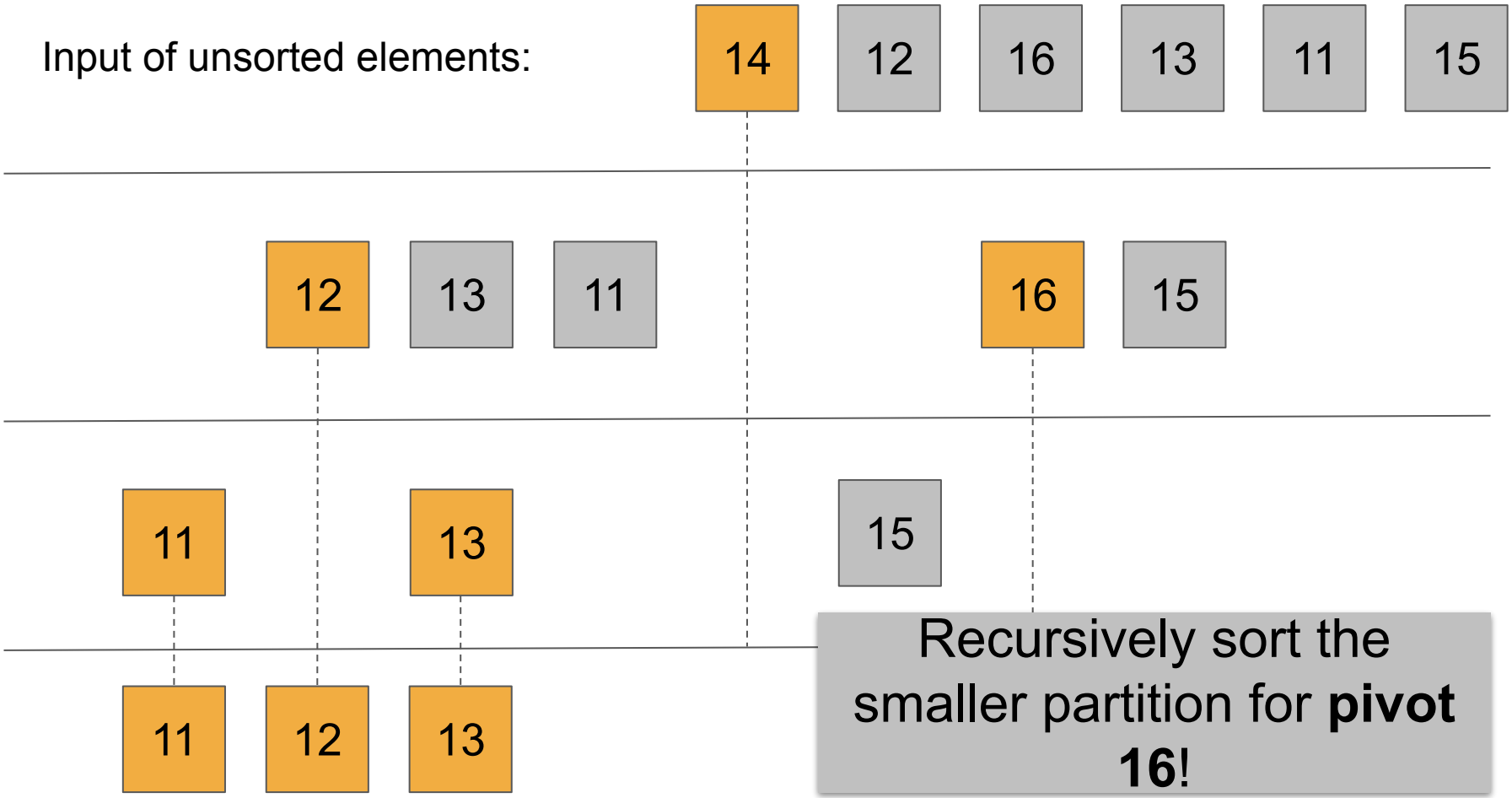

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
Now we can concatenate smaller than, equal to, and greater than for the **pivot 12**.

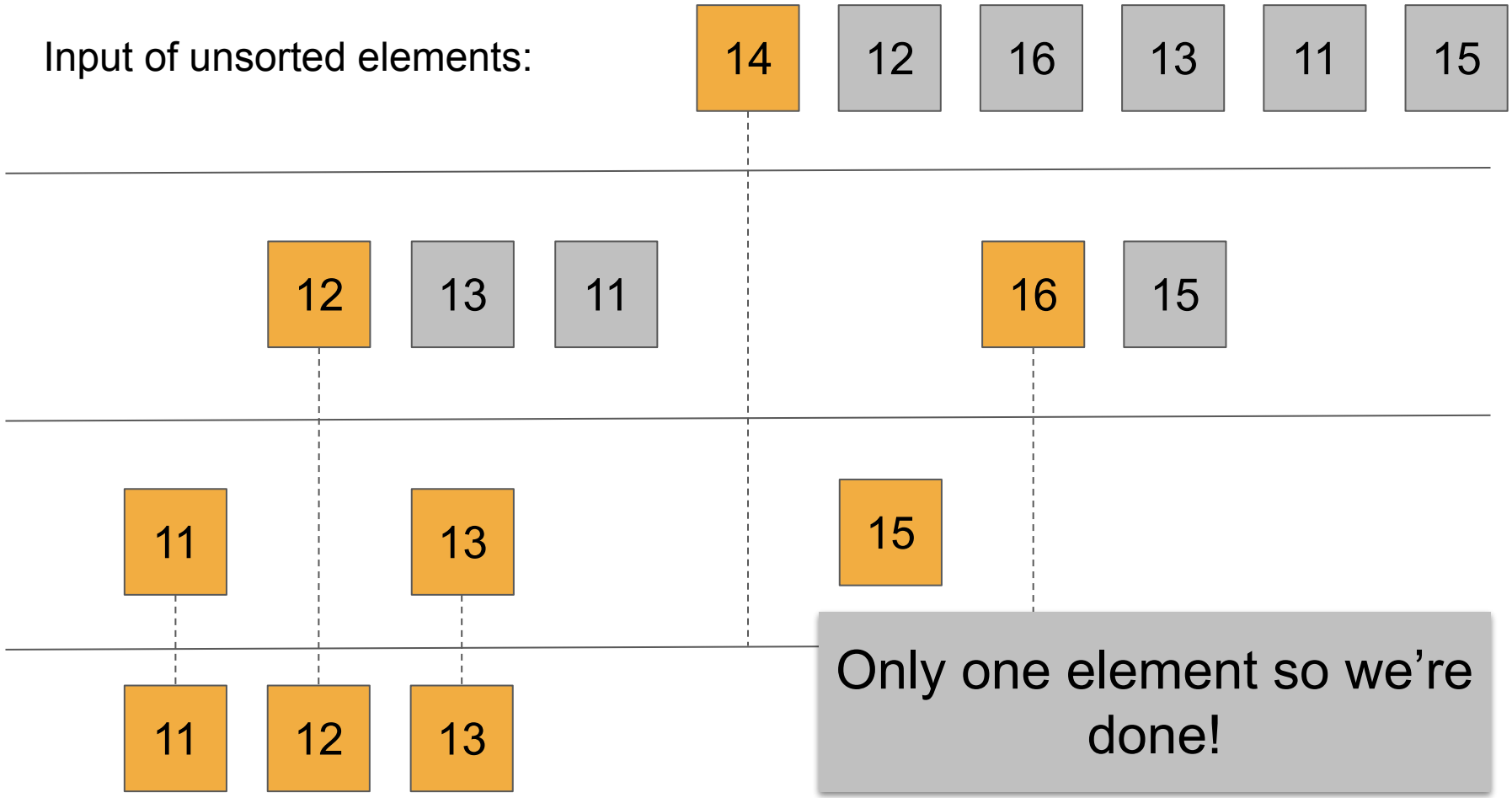

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
11
12
13

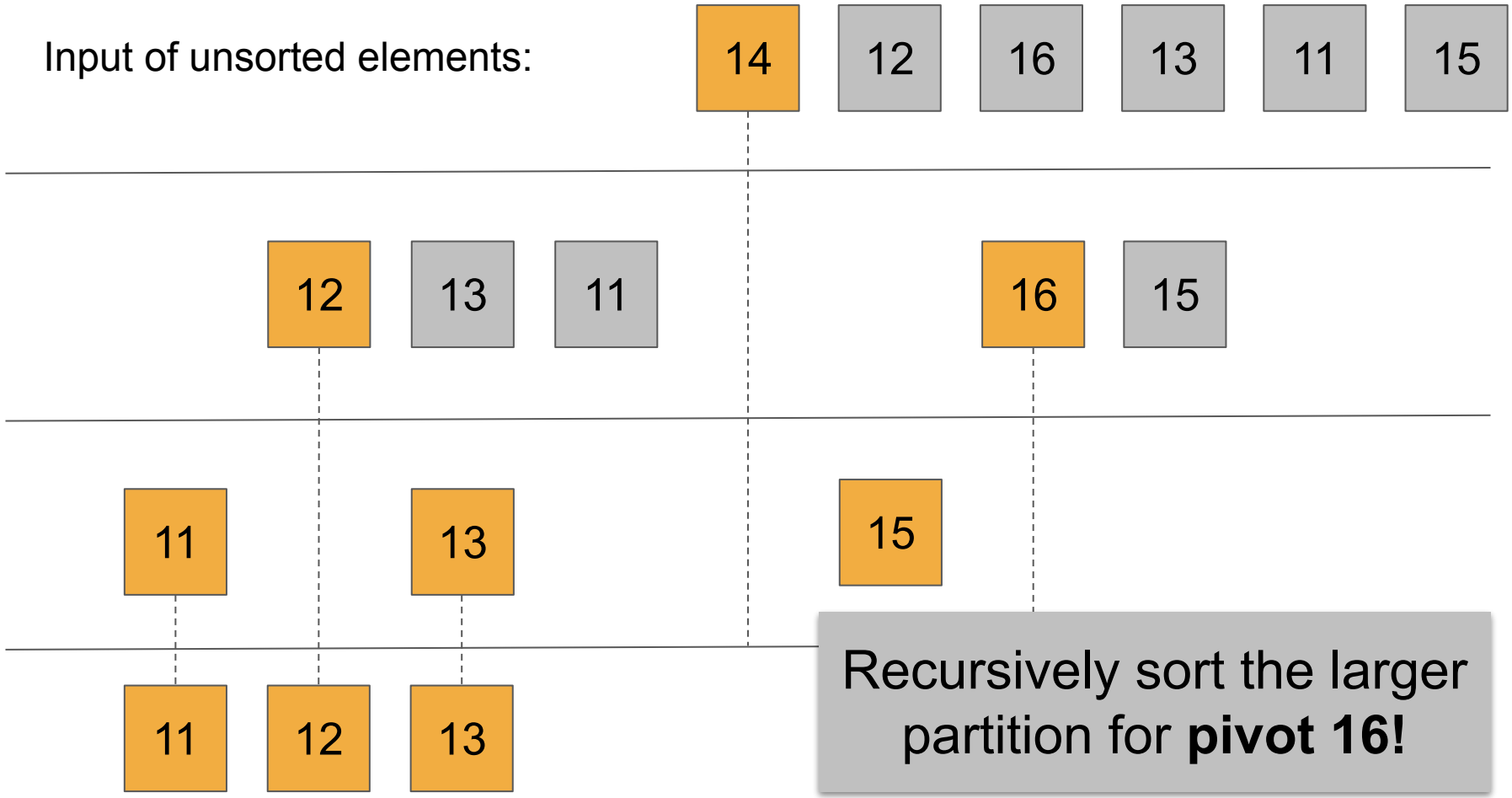

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
11
12
13
Recursively sort the larger partition for **pivot 14!**

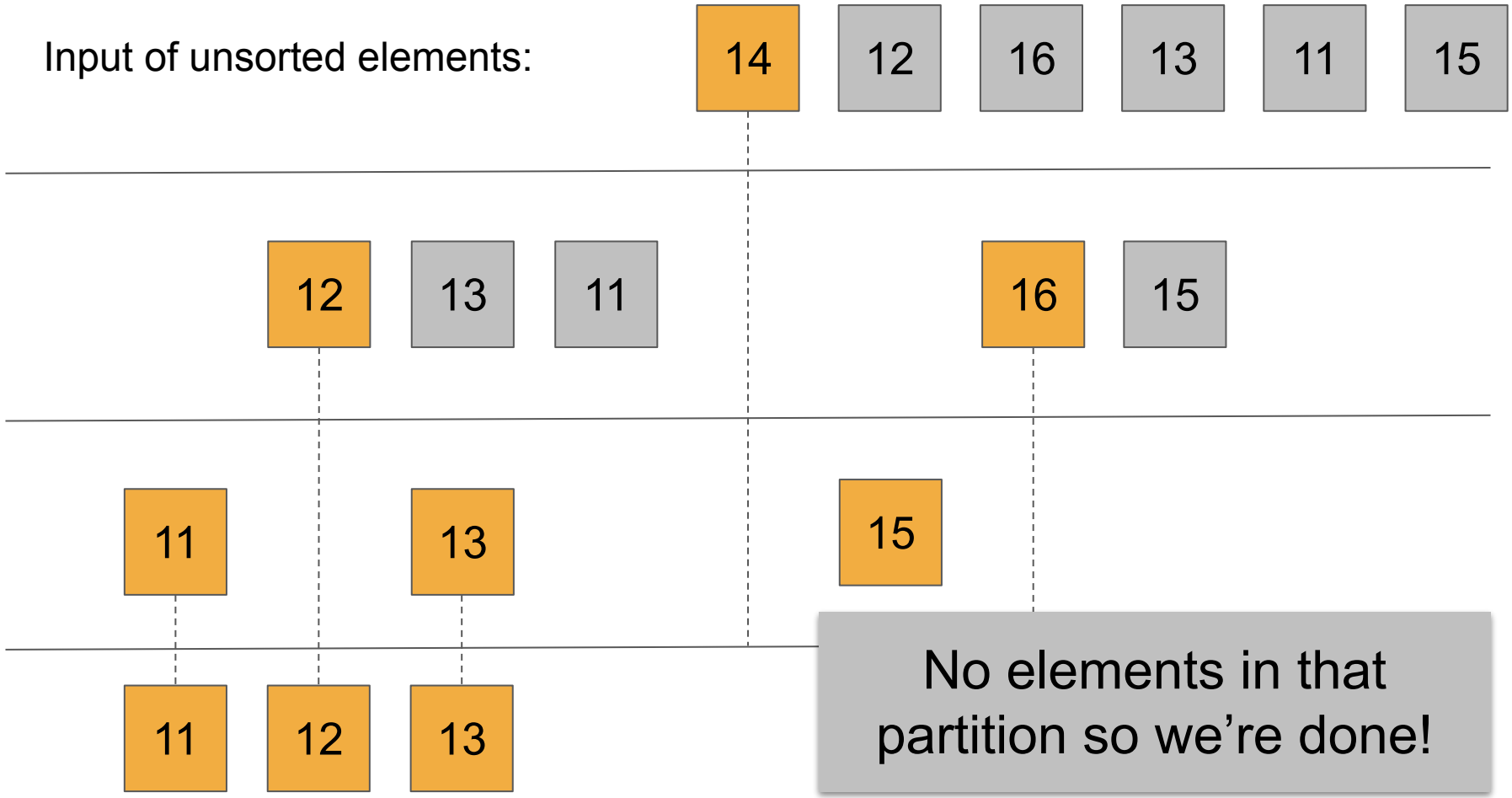

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
11
12
13
Choose the first element as the pivot.

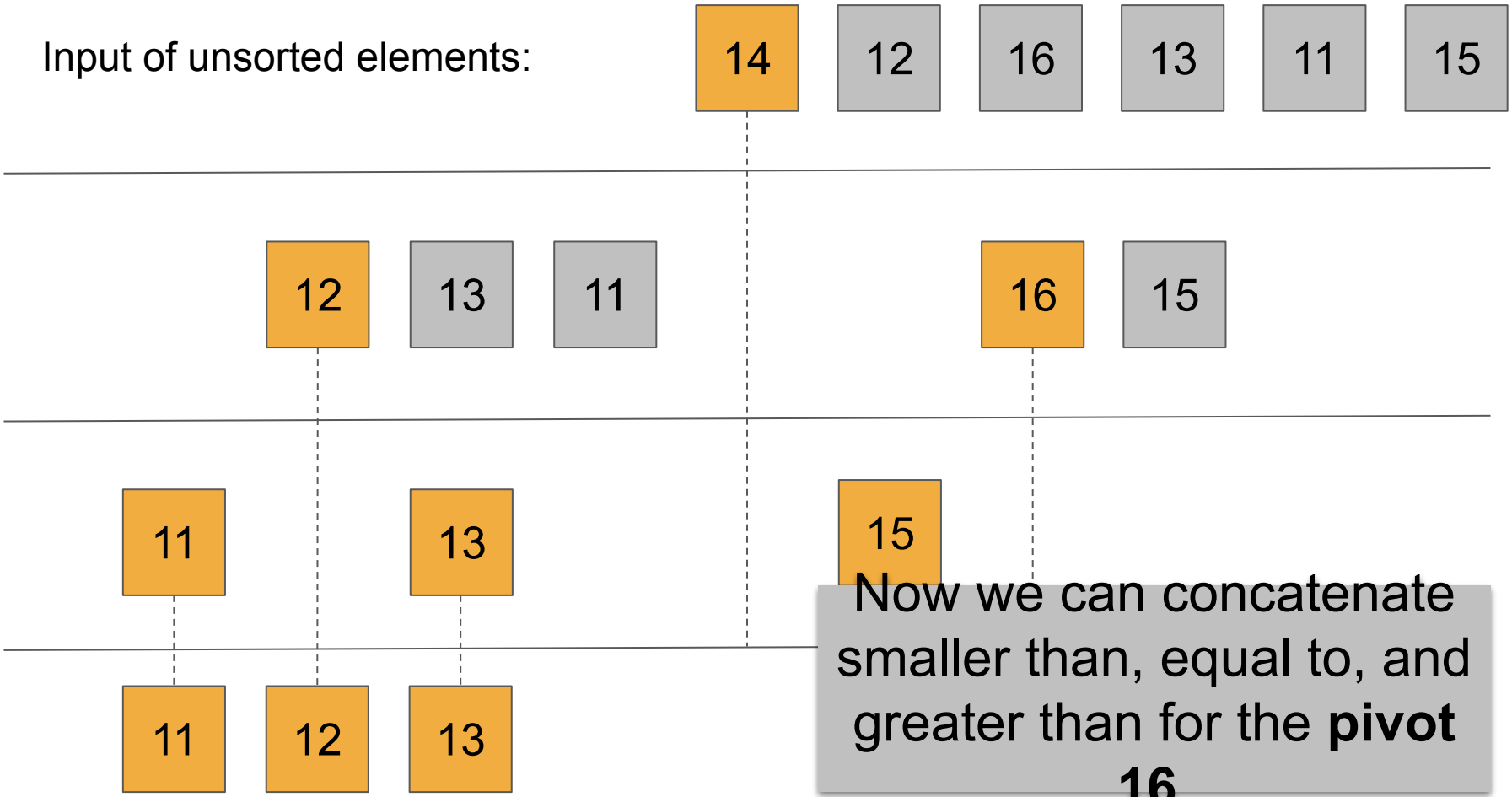

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
11
12
13
Partition elements into smaller than, equal to, and greater than the pivot.

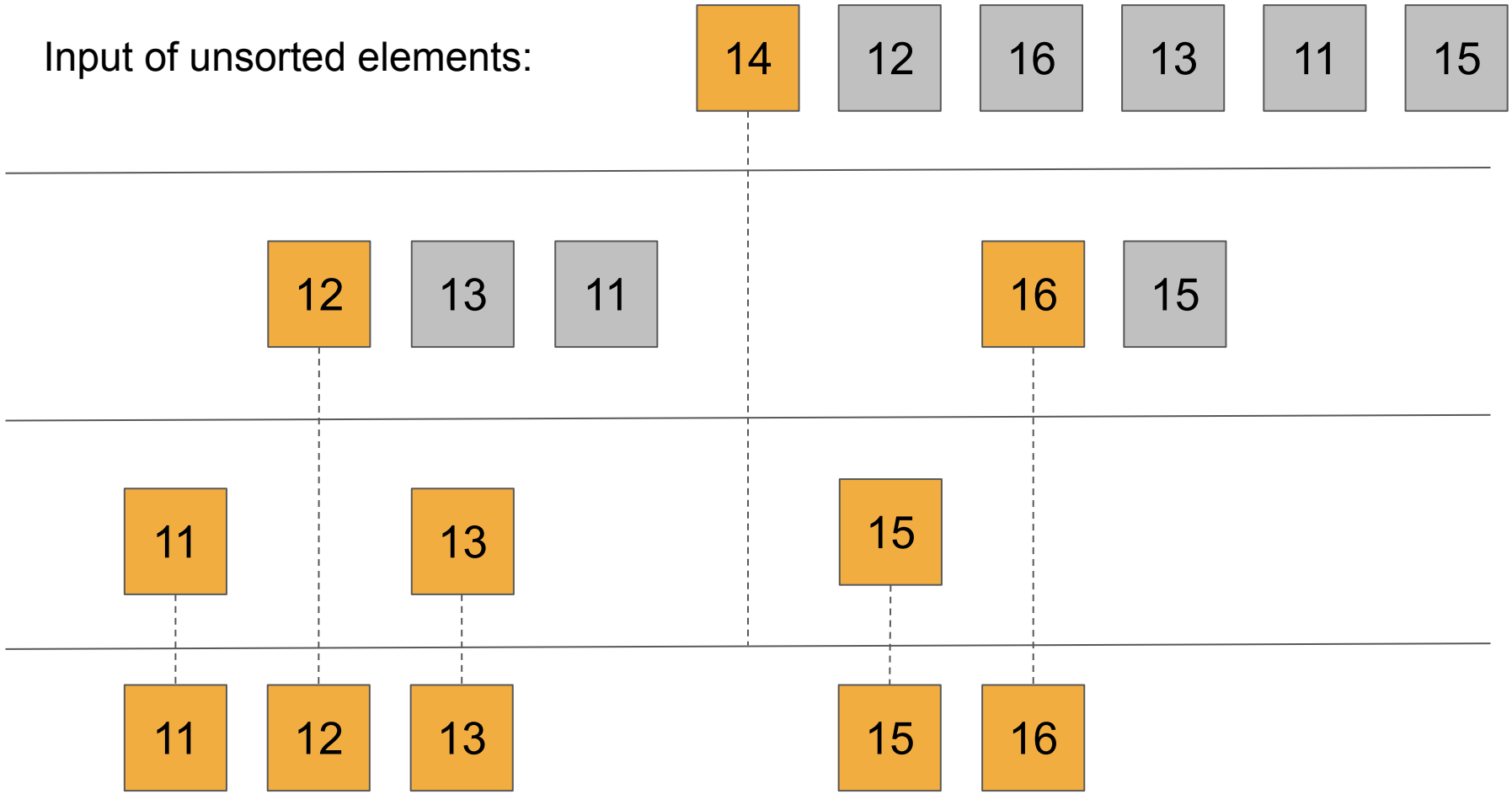

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
Recursively sort the smaller partition for **pivot 16**!

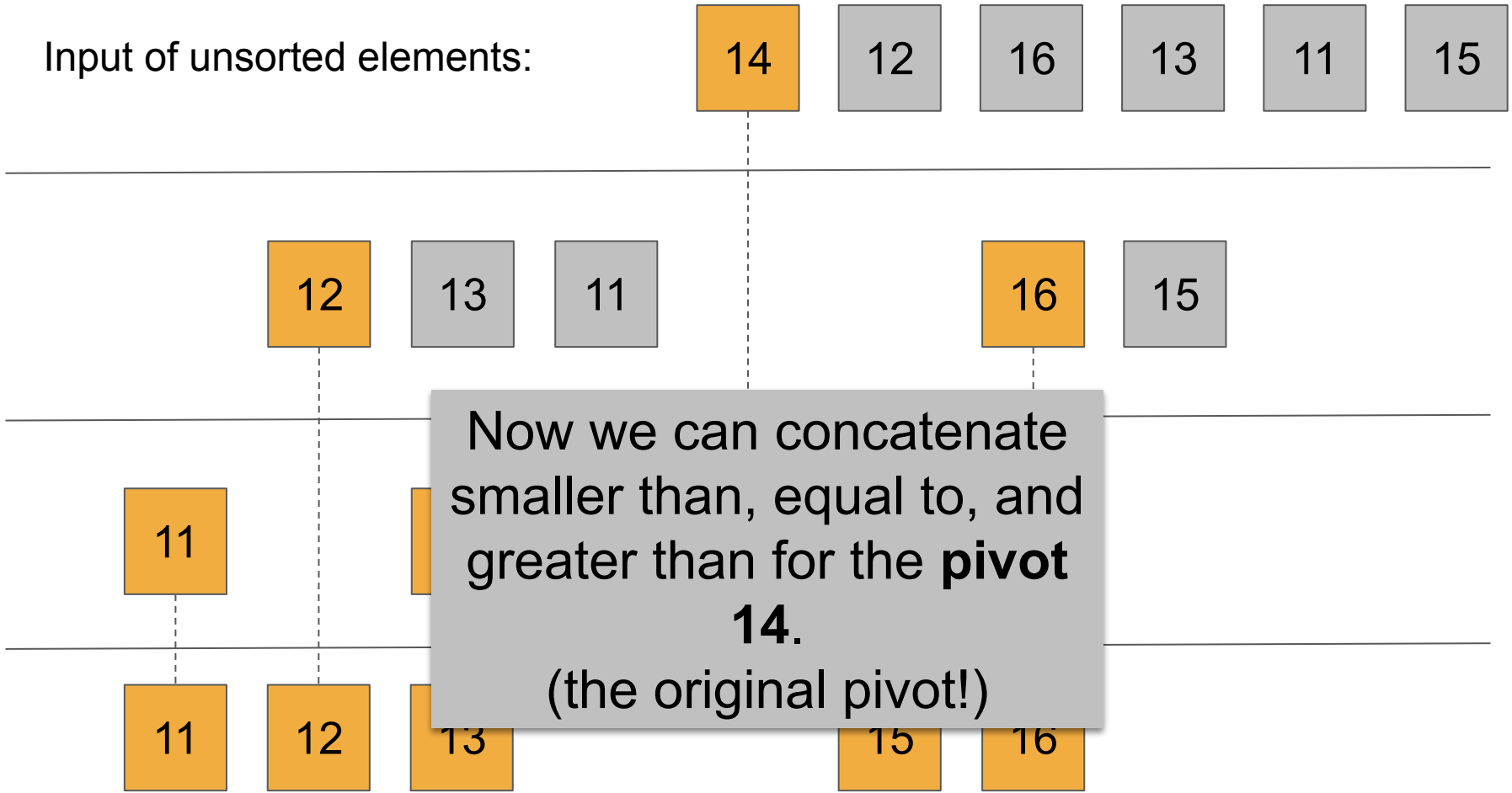

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
Only one element so we're done!

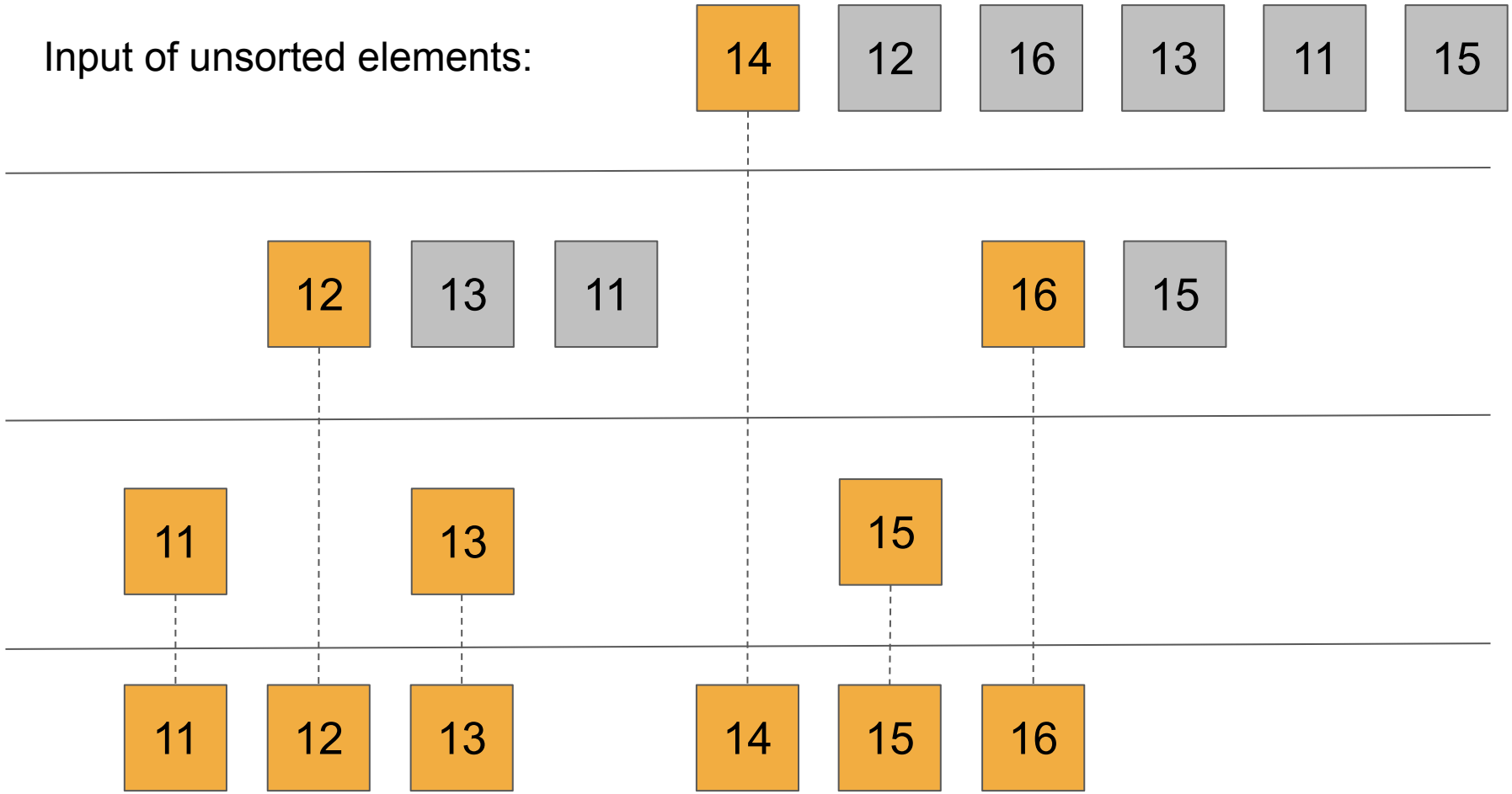

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
Recursively sort the larger partition for **pivot 16!**

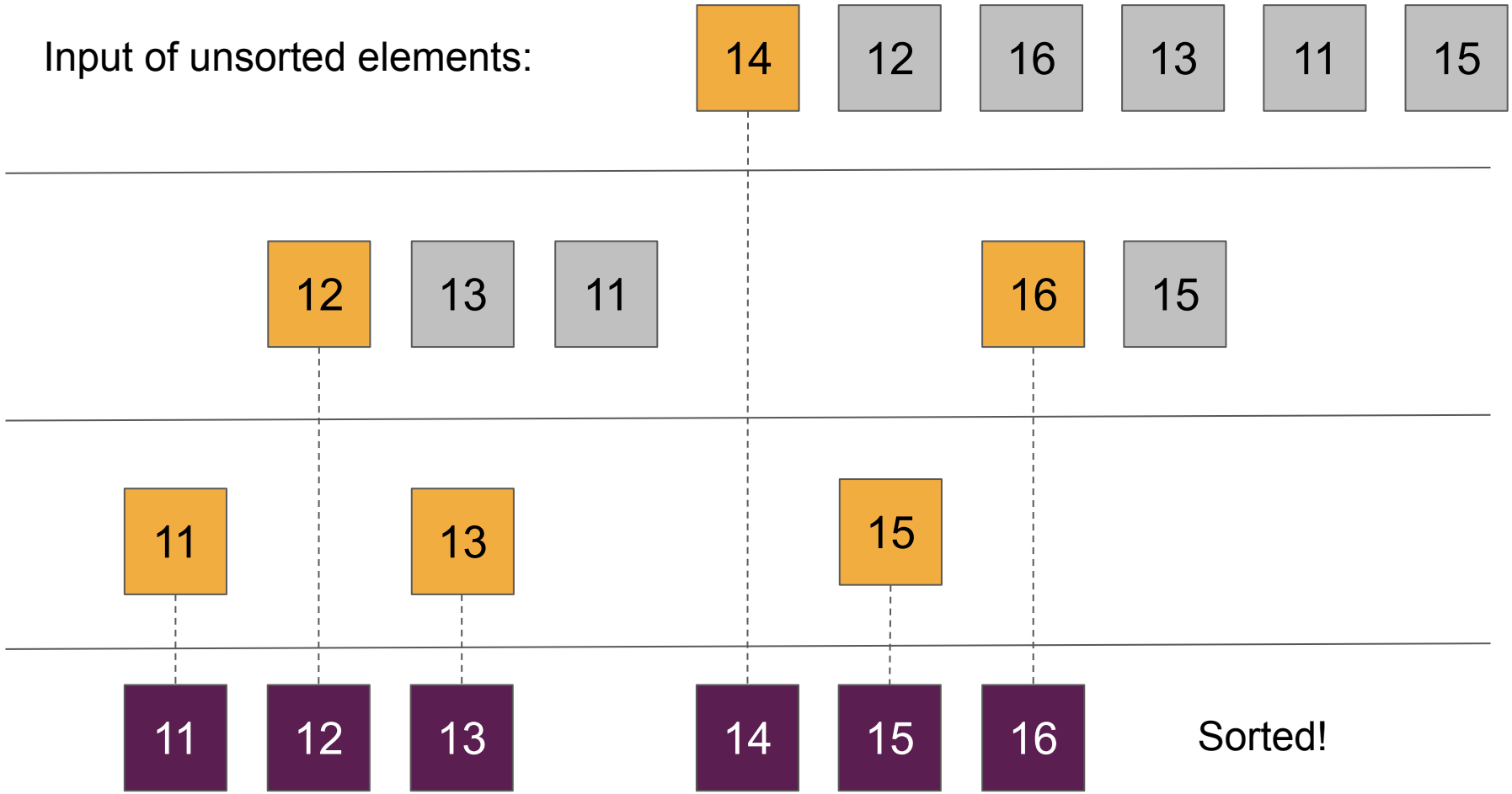

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
No elements in that partition so we're done!


Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
Now we can concatenate smaller than, equal to, and greater than for the **pivot 16**.


Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
15
16


Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
11
12
13
15
16
Now we can concatenate smaller than, equal to, and greater than for the **pivot 14**.
(the original pivot!)


Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
14
15
16


Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
14
15
16
Sorted!

Quicksort Algorithm

1. **Partition** the elements into three categories based on a chosen **pivot** element:
    - Elements **smaller** than the pivot
    - Elements **equal** to the pivot
    - Elements **larger** than the pivot

2. **Recursively sort** the two partitions that are not equal to the pivot (smaller and larger elements).
    - Now our smaller elements are in sorted order, and our larger elements are also in sorted order!

3. **Concatenate** the three now-sorted partitions together.

Quicksort –
Let's code it!

Quicksort Takeaways

- Our “divide” step = partitioning elements based on a pivot
- Our recursive call comes in between dividing and joining
  - Base case: One element or no elements to sort!
- Our “join” step = combining the sorted partitions
- Unlike in merge sort where most of the sorting work happens in the “join” step, our sorting work occurs primarily at the “divide” step for quicksort (when we sort elements into partitions).

Quicksort Efficiency Analysis

- Similar to Merge Sort, Quicksort also has `O(N log N)` runtime in the average case.
  - With good choice of pivot, we split the initial list into roughly two equally-sized parts every time.
  - Thus, we reach a depth of about `log N` split operations before reaching the base case.
  - At each level, we do `O(n)` work to partition and concatenate.

Quicksort Efficiency Analysis

- Similar to Merge Sort, Quicksort also has $O(N \log N)$ runtime in the average case.
  - With good choice of pivot, we split the initial list into roughly two equally-sized parts every time.
  - Thus, we reach a depth of about $\log N$ split operations before reaching the base case.
  - At each level, we do $O(n)$ work to partition and concatenate.

- However, Quicksort performance can degrade to $O(N^2)$ with poor choice of pivot!
  - Come talk to us after class if you're interested in why!

Quicksort Efficiency Analysis

- Similar to Merge Sort, Quicksort also has $O(N \log N)$ runtime in the average case.
  - With good choice of pivot, we split the initial list into roughly two equally-sized parts every time.
  - Thus, we reach a depth of about $\log N$ split operations before reaching the base case.
  - At each level, we do $O(n)$ work to partition and concatenate.

- However, Quicksort performance can degrade to $O(N^2)$ with poor choice of pivot!
  - Come talk to us after class if you're interested in why!

- The ultimate question: Can we do better?
  - From a space efficiency perspective, yes, there are versions of Quicksort that don't require making many copies of the list (in-place Quicksort). But from a runtime efficiency perspective...

The Limit Does Exist

- There is a **fundamental limit** on the efficiency of comparison-based sorting algorithms.

The Limit Does Exist

- There is a **fundamental limit** on the efficiency of comparison-based sorting algorithms.

- You can prove that it is not possible to guarantee a list has been sorted unless you have done **at minimum `O(N log N)` comparisons**.

The Limit Does Exist

- There is a **fundamental limit** on the efficiency of comparison-based sorting algorithms.

- You can prove that it is not possible to guarantee a list has been sorted unless you have done **at minimum `O(N log N)` comparisons**.

- Thus, we can't do better (in Big-O terms at least) than Merge Sort and Quicksort!

Final Advice

Assignment 5 Tips

- When implementing the two sorting algorithms, it is strongly recommended to implement helper functions for the divide/join components of the algorithm.
  - For merge sort this means having helper functions for the split and merge operations
  - For quicksort this means having helper functions for the partition and concatenate operations

Assignment 5 Tips

- When implementing the two sorting algorithms, it is strongly recommended to implement helper functions for the divide/join components of the algorithm.
  - For merge sort this means having helper functions for the split and merge operations
  - For quicksort this means having helper functions for the partition and concatenate operations

- These helper functions should be implemented iteratively, but the overall sorting algorithms themselves operate recursively. Mind the distinction!

Assignment 5 Tips

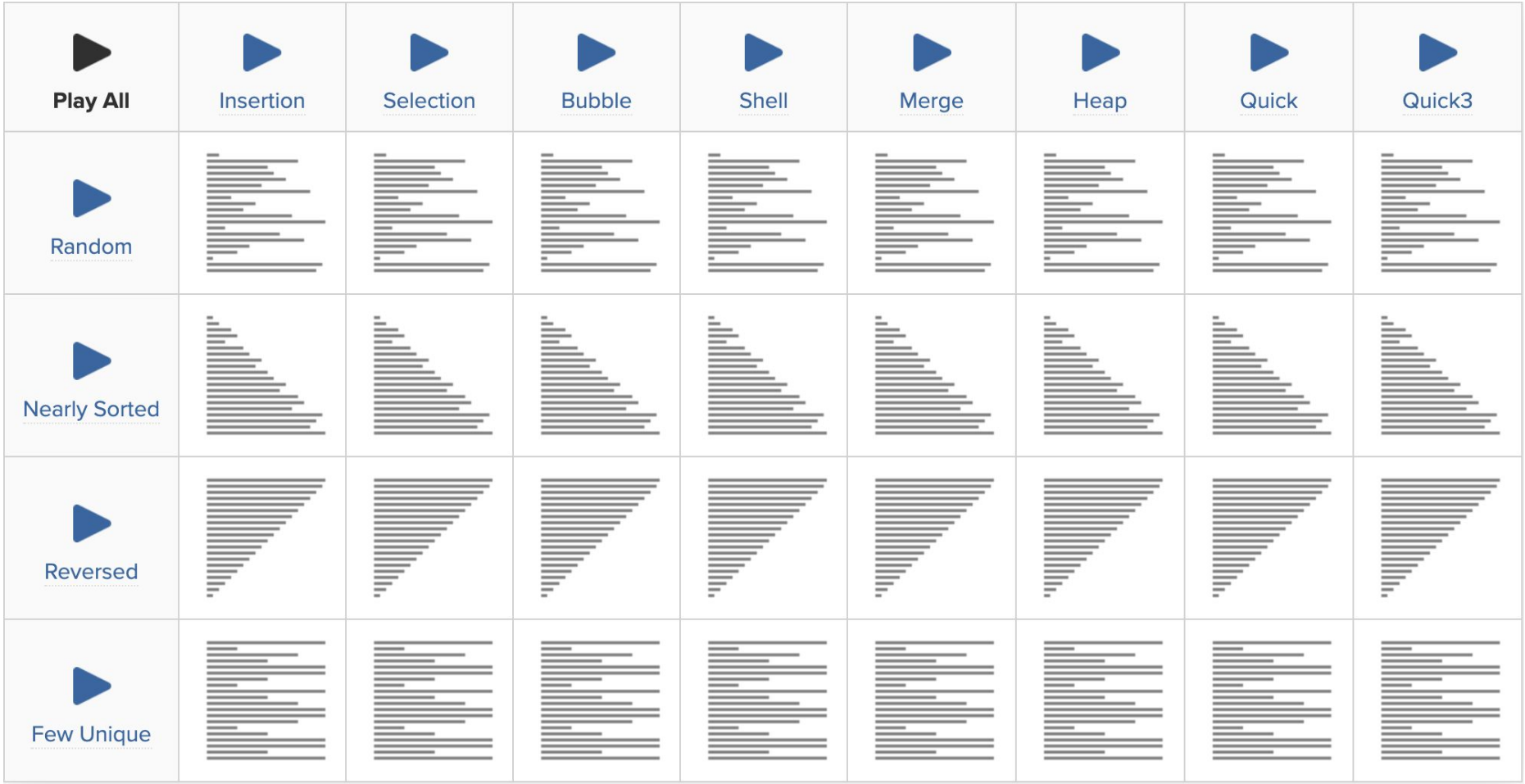
- When implementing the two sorting algorithms, it is strongly recommended to implement helper functions for the divide/join components of the algorithm.
  - For merge sort this means having helper functions for the split and merge operations
  - For quicksort this means having helper functions for the partition and concatenate operations

- These helper functions should be implemented iteratively, but the overall sorting algorithms themselves operate recursively. Mind the distinction!

- Write tests for your helper functions first! Then, write end-to-end tests for your sorting functions.

Summary

https://www.toptal.com/developers/sorting-algorithms



Sorting Big-O Cheat Sheet			
Sort	Worst Case	Best Case	Average Case
Insertion	O(n^2)	O(n)	O(n^2)
Selection	O(n^2)	O(n^2)	O(n^2)
Merge	O(n log n)	O(n log n)	O(n log n)
Quicksort	O(n^2)	O(n log n)	O(n log n)

Sorting

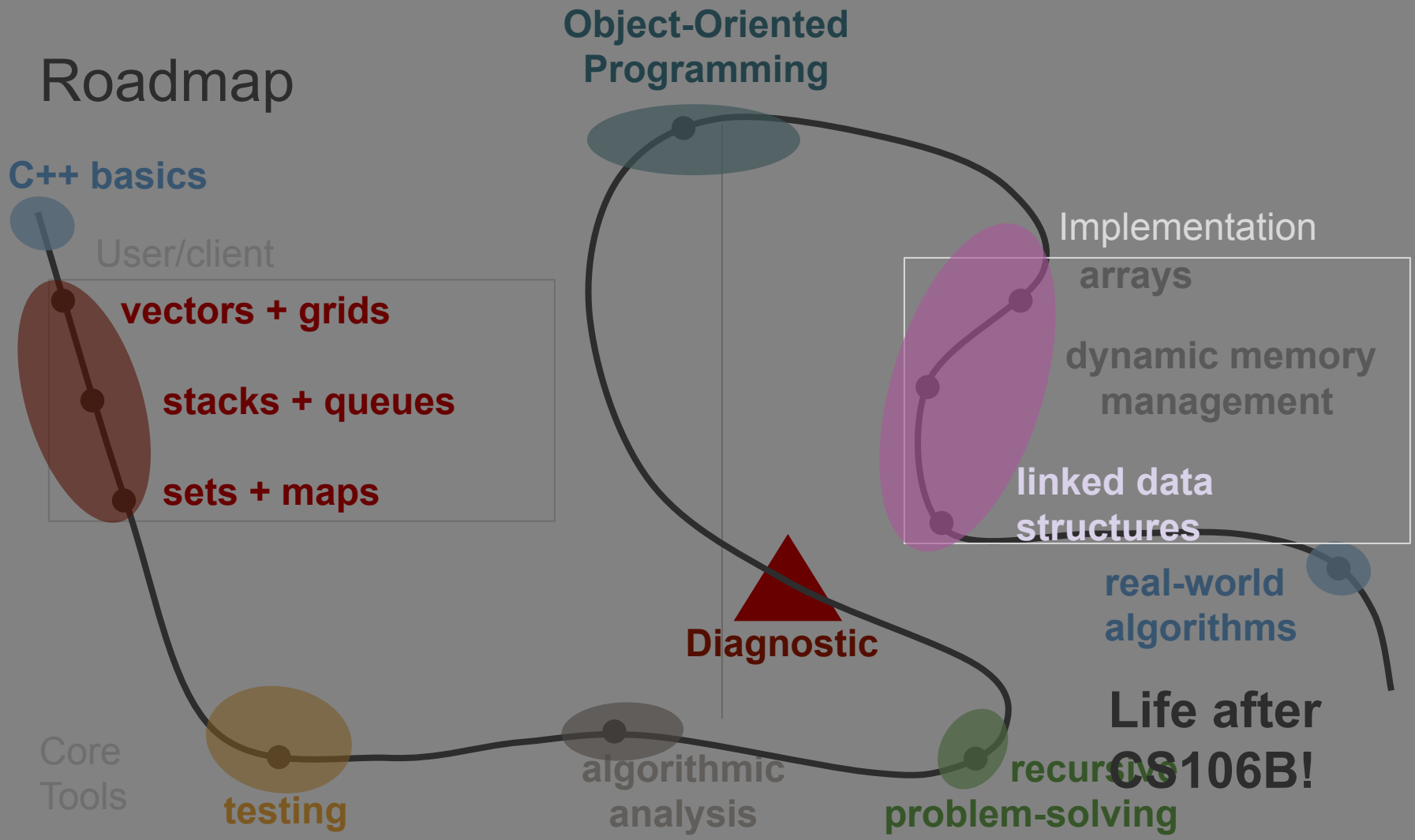
- Sorting is a powerful tool for organizing data in a meaningful format!

- There are many different methods for sorting data:
  - Selection Sort
  - Insertion Sort
  - Mergesort
  - Quicksort
  - And many more…

- Understanding the different runtimes and tradeoffs of the different algorithms is important when choosing the right tool for the job!

What’s next?

Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Trees!