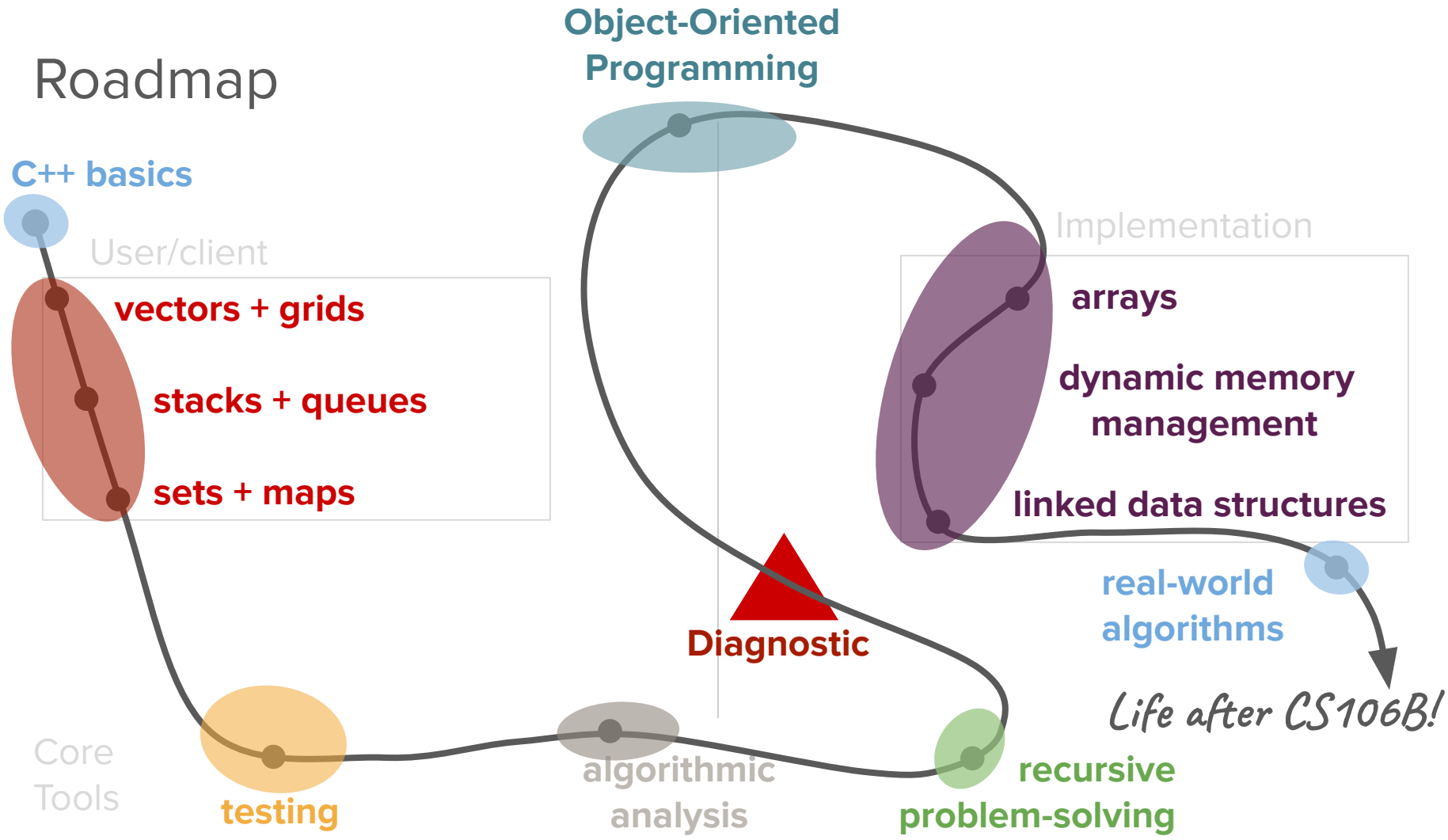
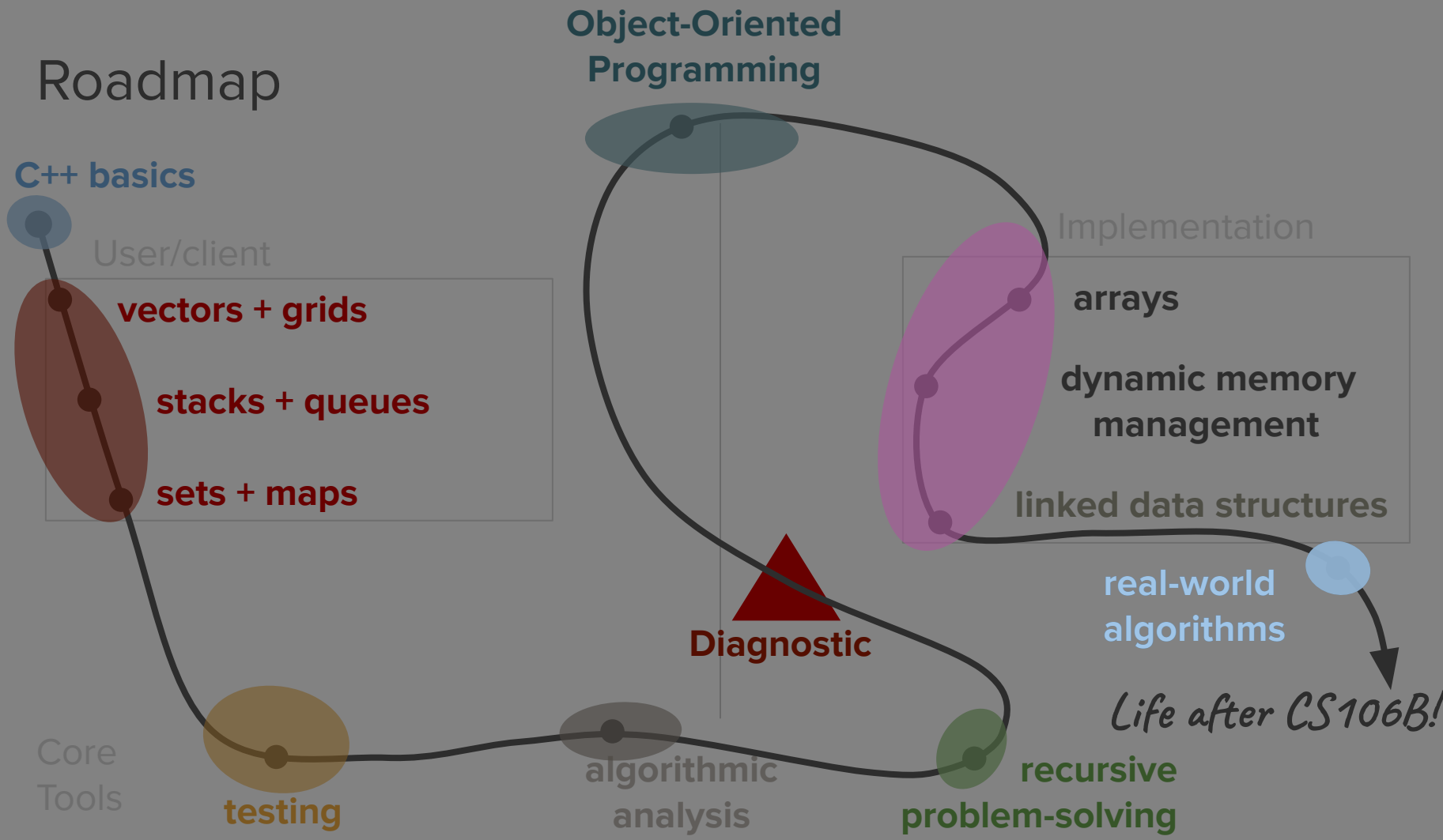
Sorting Algorithms

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

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

What are some real-world algorithms that can be used to organize data?

How can we design better, more efficient sorting algorithms?

Today's topics

1. Review
2. Introduction to Sorting
3. Selection Sort
4. Divide-and-Conquer Sorts (MergeSort and QuickSort)

Review

[linked list operations]

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?

Linked List Traversal Takeaways

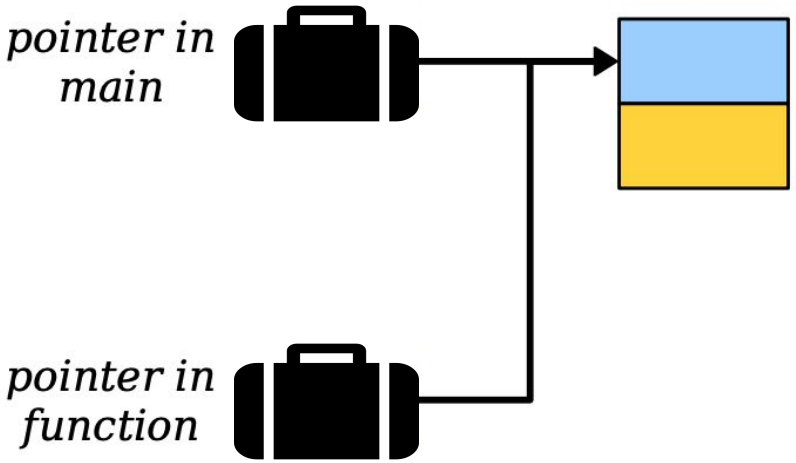
- Temporary pointers into lists are very helpful!
  - When processing linked lists iteratively, it's common to introduce pointers that point to cells in multiple spots in the list.
  - This is particularly useful if we're destroying or rewiring existing lists.

- Using a **`while`** loop with a condition that checks to see if the current pointer is **`nullptr`** is the prevailing way to traverse a linked list.

- Iterative traversal offers the most flexible, scalable way to write utility functions that are able to handle all different sizes of linked lists.

Pointers by Value

- Unless specified otherwise, function arguments in C++ are passed by value – this includes pointers!
- A function that takes a pointer as an argument gets a copy of the pointer.
- We can change where the copy points, but not where the original pointer points.



Pointers by Reference

- To solve this problem, we can **pass the linked list pointer by reference.**

- The mechanics of how to do so:

```
void prependTo(Node*& list, string data) {
    Node* newNode = new Node;
    newNode->data = data;

    newNode->next = list;
    list = newNode;
}
```

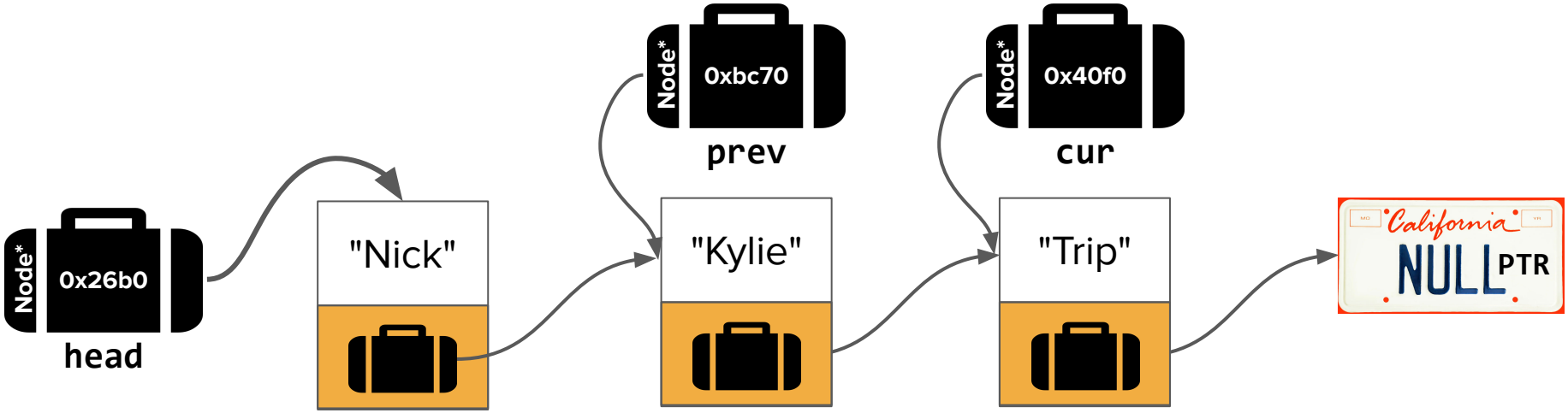
This is a reference to a pointer to a Node. If we change where list points in this function, the changes will stick!

A more efficient **appendTo()** - using a tail pointer!

```
Node* createListWithTailPtr(Vector<string> values) {
    if (values.isEmpty()) return nullptr;
    Node* head = new Node(values[0], nullptr);

    Node* cur = head;
    for (int i = 1; i < values.size(); i++) {
        Node* newNode = new Node(values[i], nullptr);
        cur->next = newNode;
        cur = newNode;
    }
    return head;
}
```

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

- We saw lots of ways to examine and manipulate linked lists!
  - Traversal
  - Rewiring
  - Insertion (front/back/middle)
  - Deletion (front/back/middle)

- We saw linked lists in classes and outside classes, and pointers passed by value and passed by reference.

- Assignment 6 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

What are some real-world algorithms that can be used to organize data?

What is sorting?

This is one kind of sorting...

This is one kind of sorting...

This is one kind of sorting... but not quite what we mean!

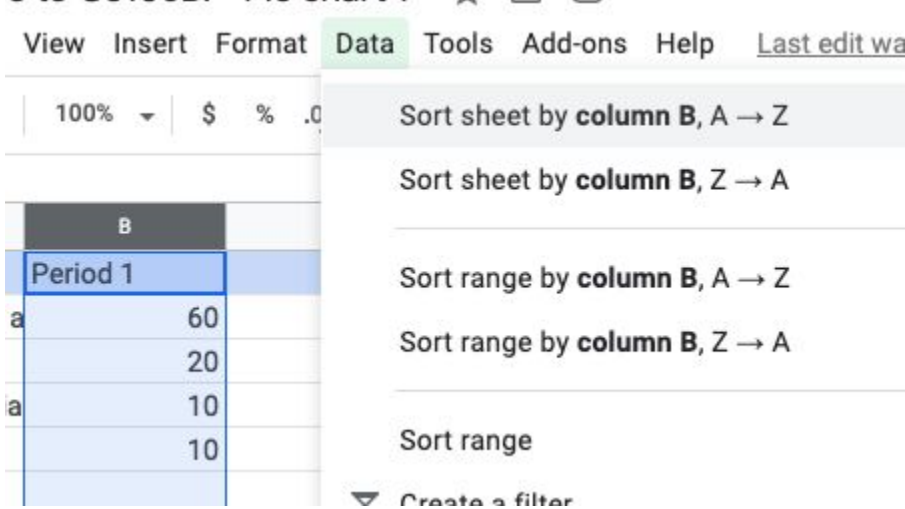
Definition

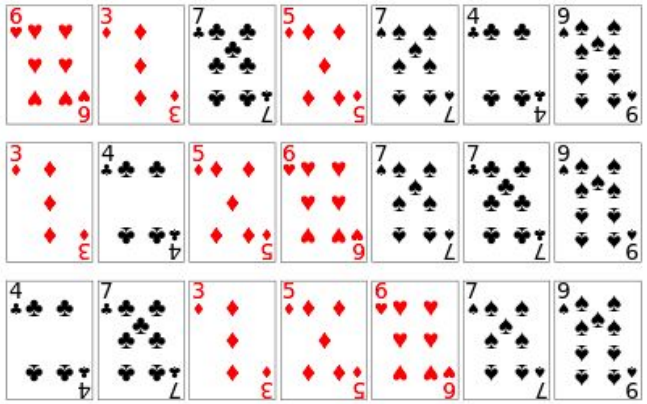
sorting

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

Why is sorting useful?

Time	Auto	Athlete	Nationality	Date	Venue
4:37.0		Anne Smith	United Kingdom	3 June 1967[8]	London
4:36.8		Maria Gommers	Netherlands	14 June 1969[8]	Leicester
4:35.3		Ellen Tittel	West Germany	20 August 1971[8]	Sittard
4:29.5		Paola Pigni	Italy	8 August 1973[8]	Viareggio
4:23.8		Natalia Mărășescu	Romania	21 May 1977[8]	Bucharest
4:22.1	4:22.09	Natalia Mărășescu	Romania	27 January 1979[8]	Auckland
4:21.7	4:21.68	Mary Decker	United States	26 January 1980[8]	Auckland
	4:20.89	Lyudmila Veselkova	Soviet Union	12 September 1981[8]	Bologna
	4:18.08	Mary Decker-Tabb	United States	9 July 1982[8]	Paris
	4:17.44	Maricica Puică	Romania	9 September 1982[8]	Rieti
	4:16.71	Mary Decker-Slaney	United States	21 August 1985[8]	Zürich
	4:15.61	Paula Ivan	Romania	10 July 1989[8]	Nice
	4:12.56	Svetlana Masterkova	Russia	14 August 1996[8]	Zürich
	4:12.33	Sifan Hassan	Netherlands	12 July 2019	Monaco





Approaches to sorting

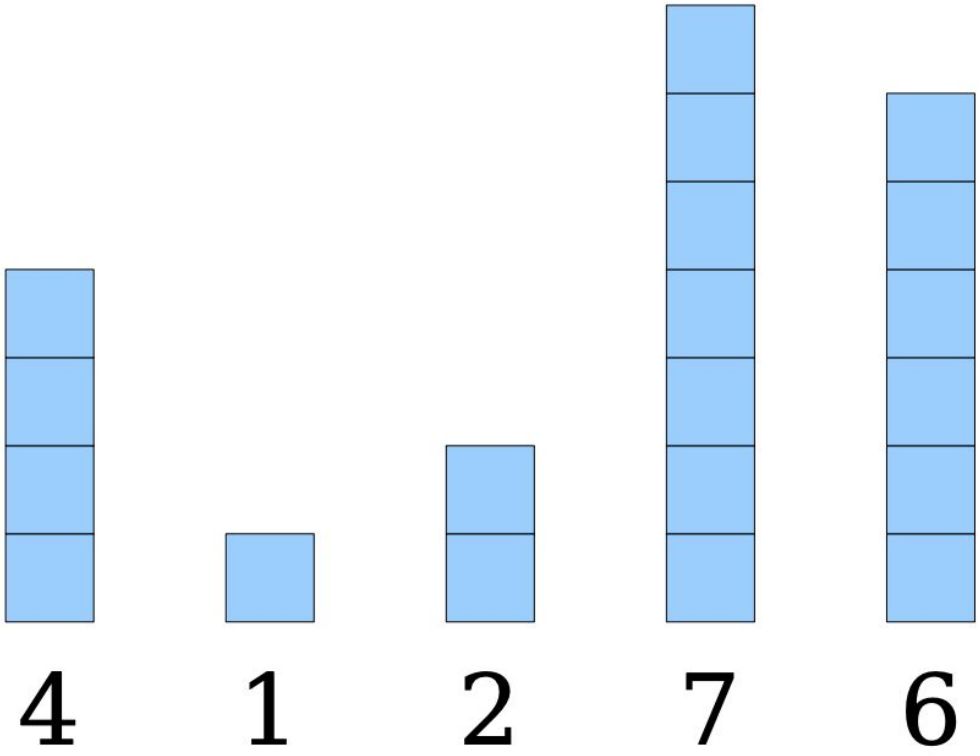
- Suppose we want to rearrange a sequence to put elements into ascending order (each element is less than or equal to the element that follows it).

- In this lecture, we're going to answer the following questions:
  - What are some strategies we could use?
  - How do those strategies compare?
  - Is there a “best” strategy?

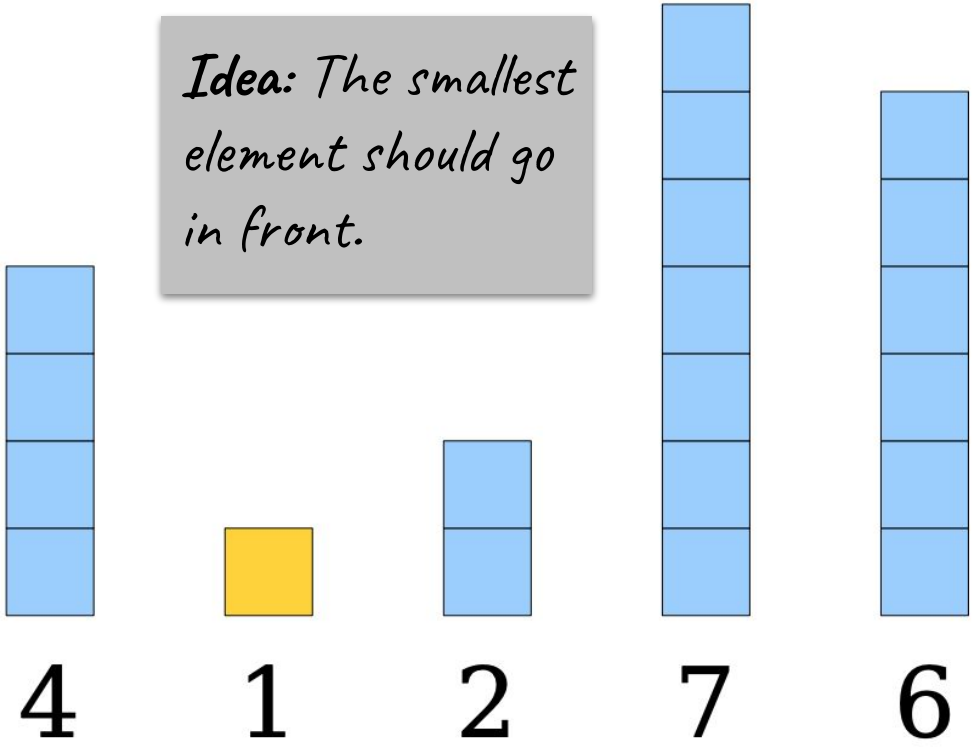
Sorting algorithms

Animations courtesy of Keith Schwarz!

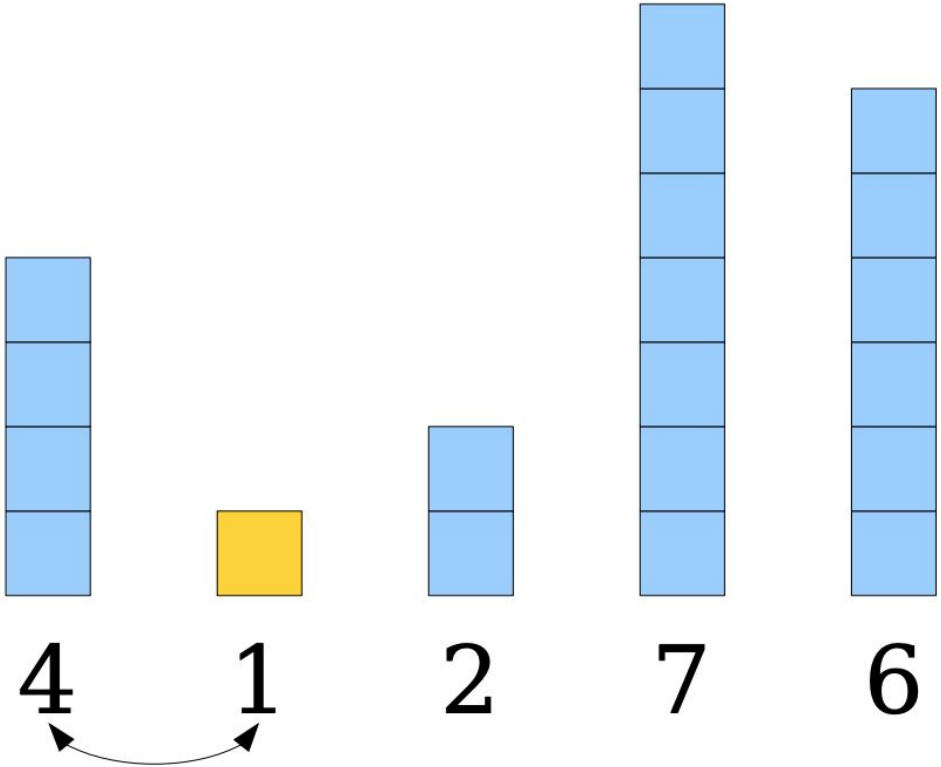
Our first sort: Selection sort



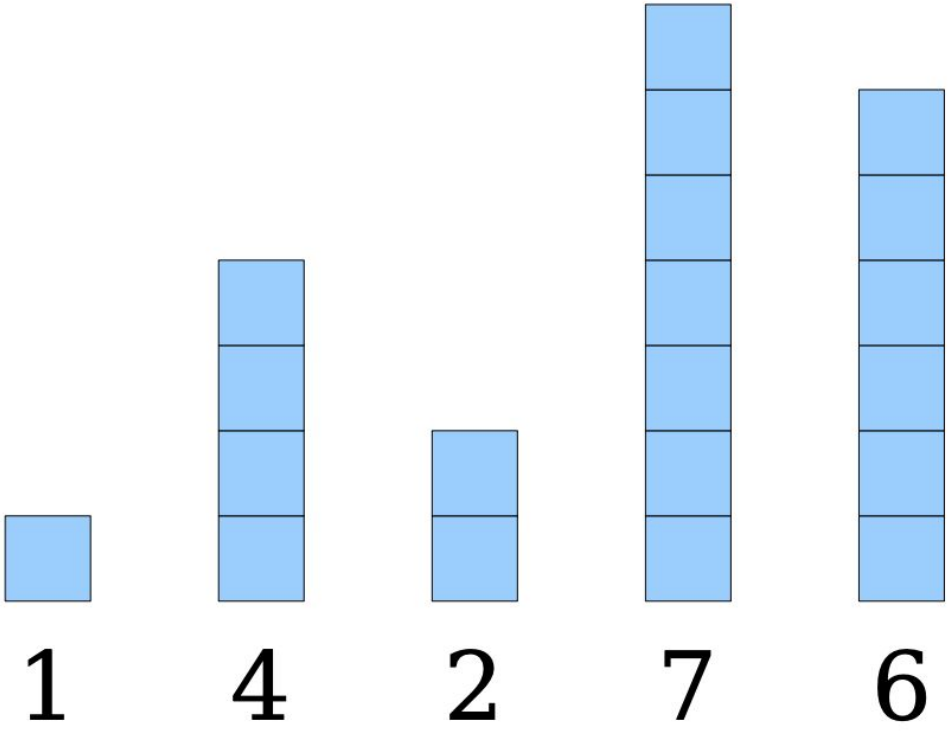
Our first sort: Selection sort



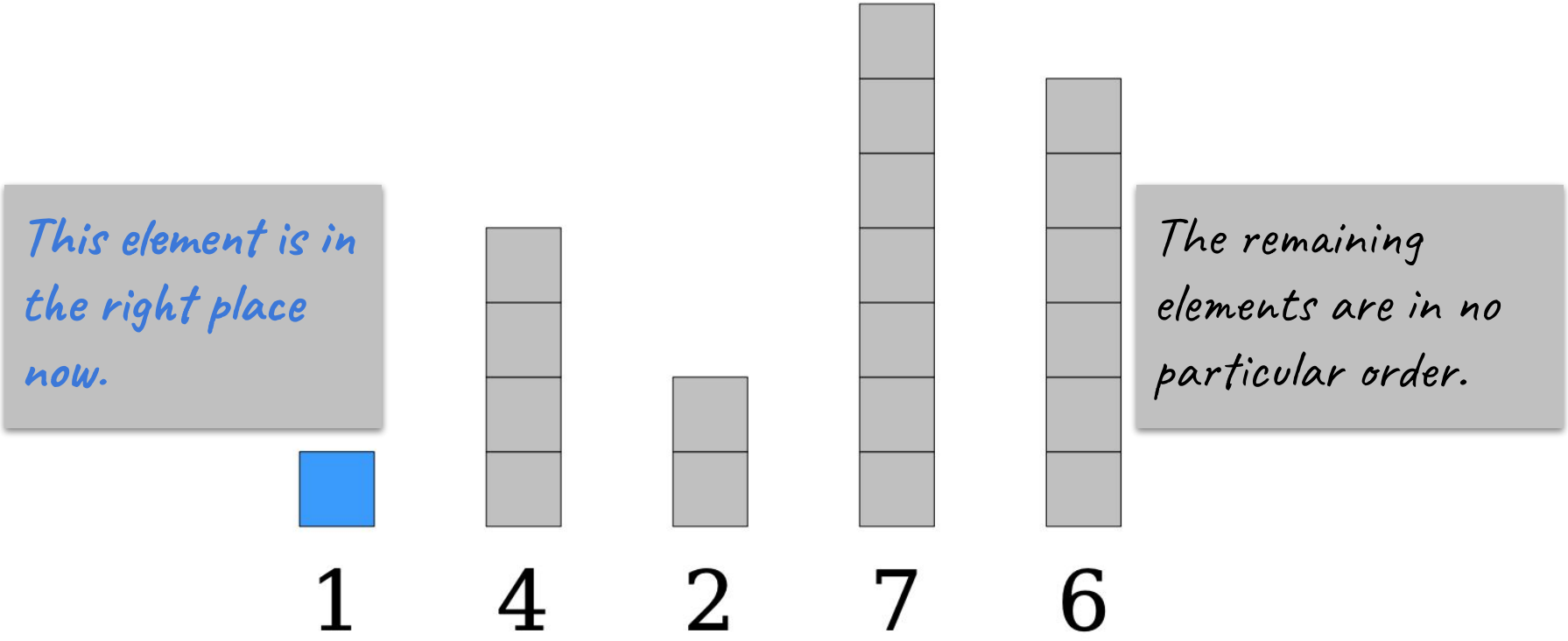
Our first sort: Selection sort



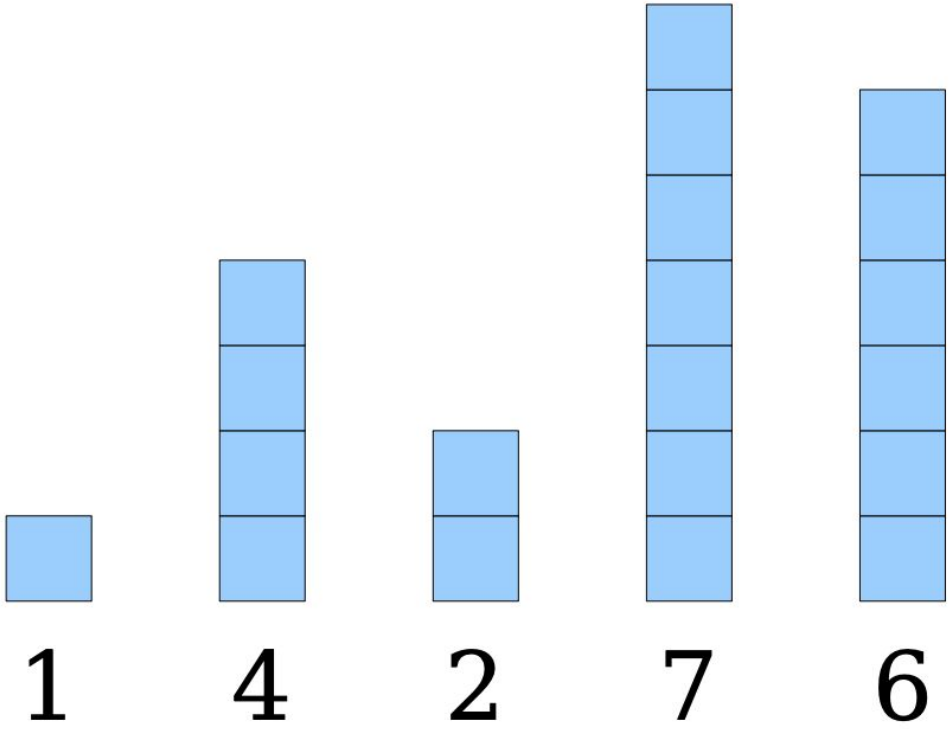
Our first sort: Selection sort



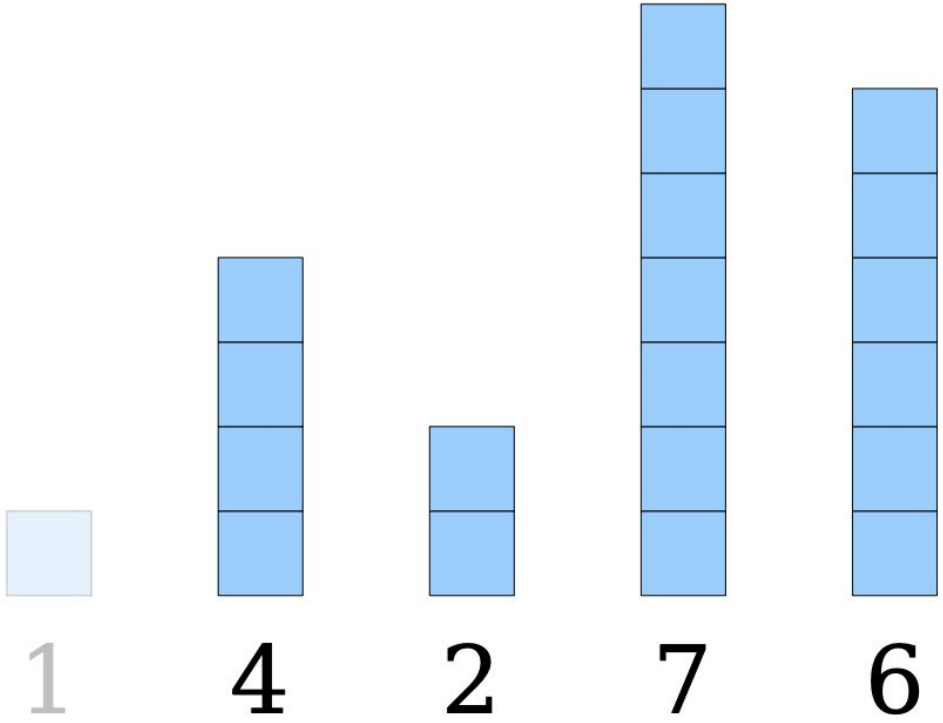
Our first sort: Selection sort



Our first sort: Selection sort

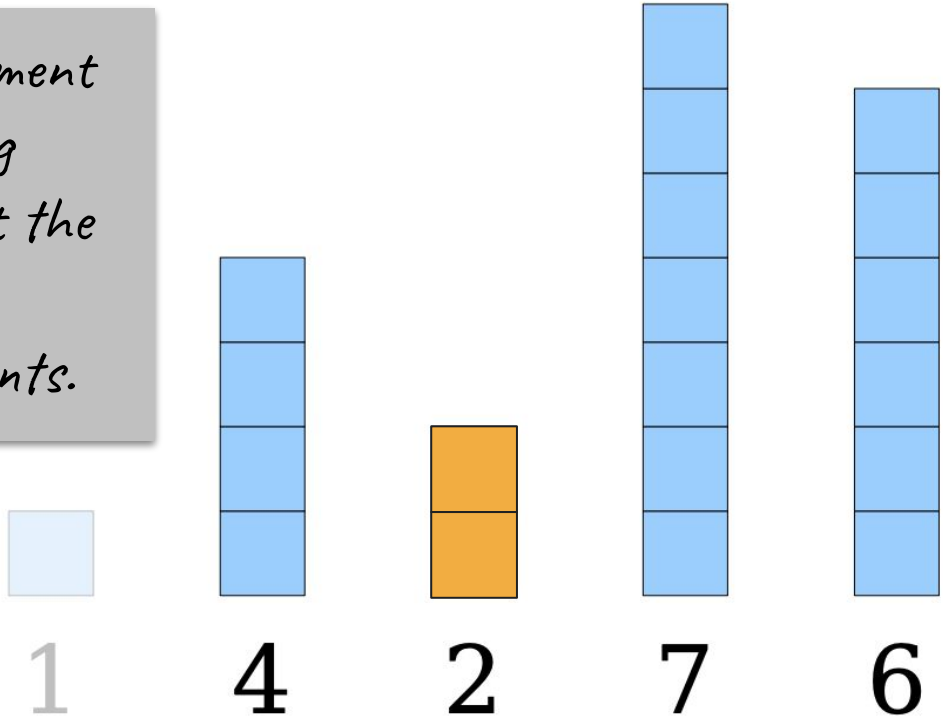


Our first sort: Selection sort

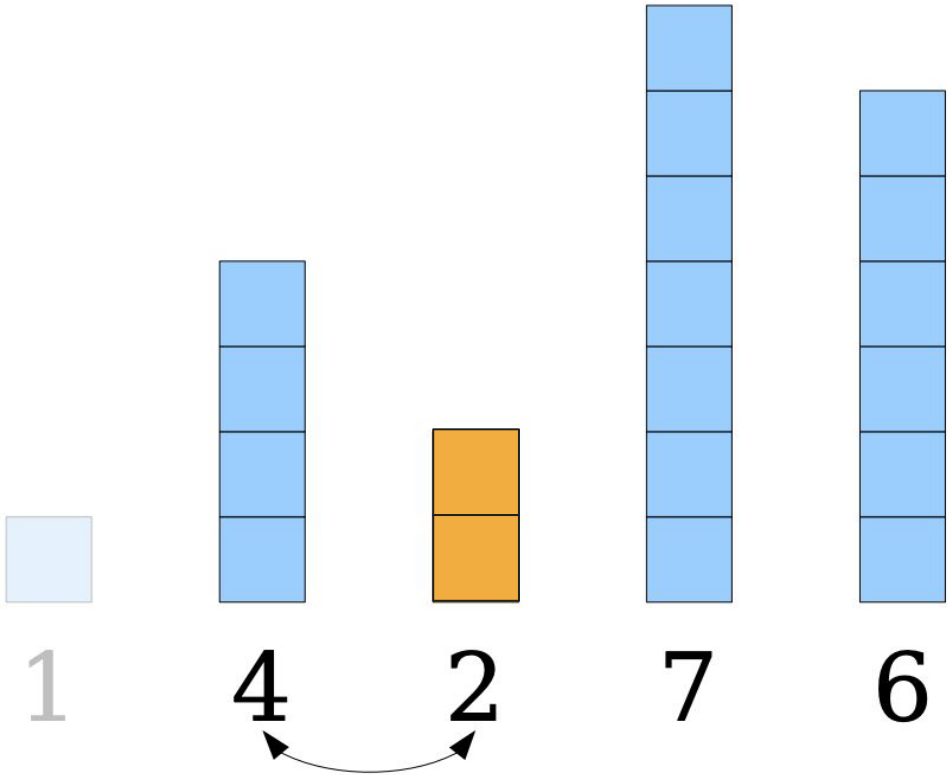


Our first sort: Selection sort

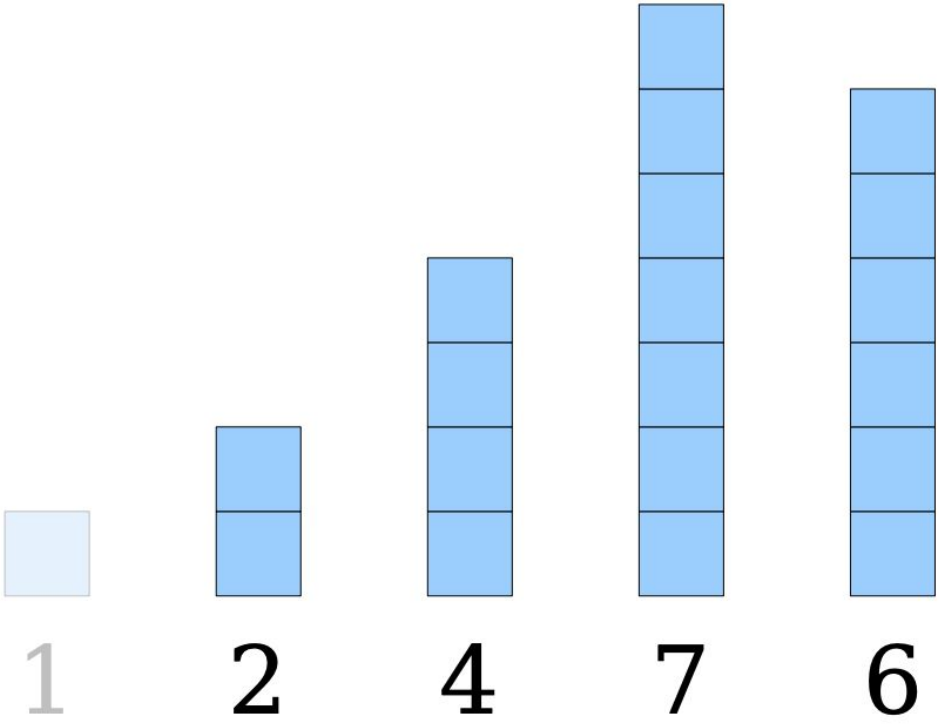
The smallest element of the remaining elements goes at the front of the remaining elements.



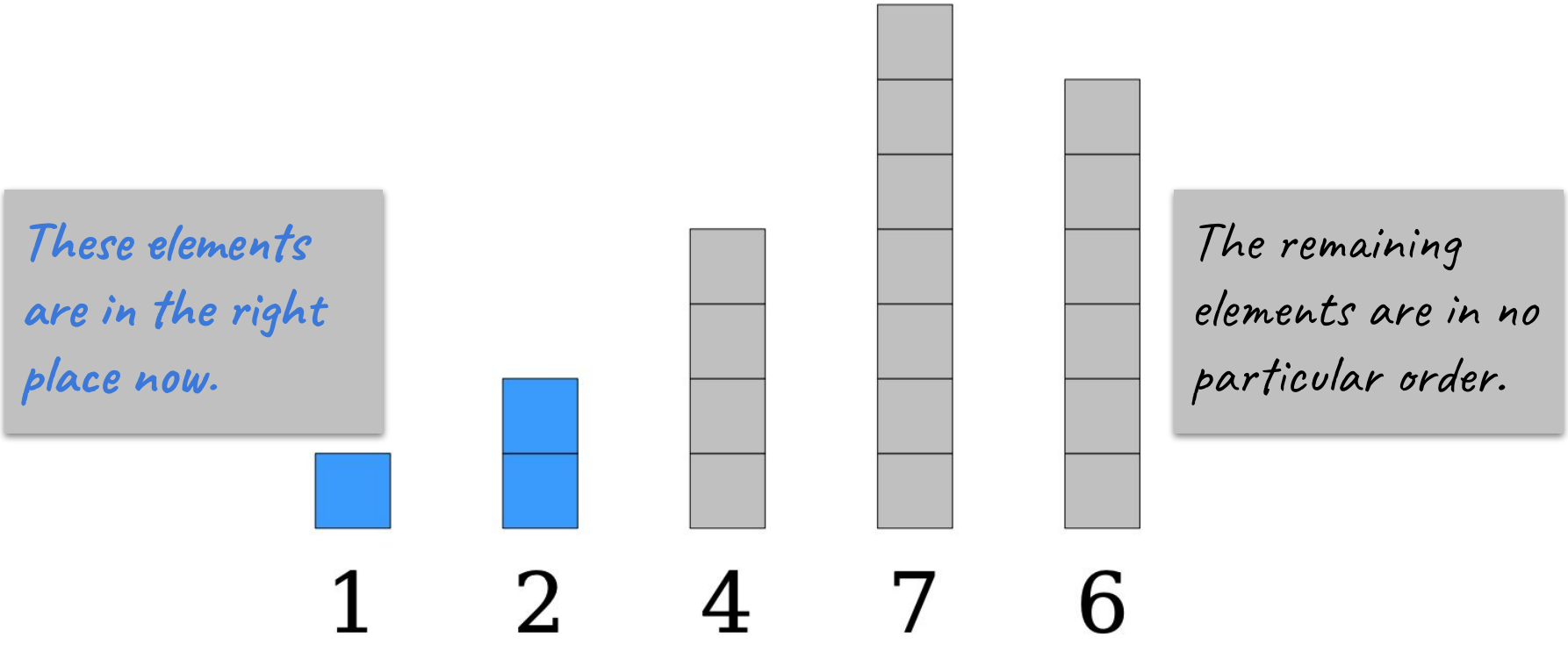
Our first sort: Selection sort



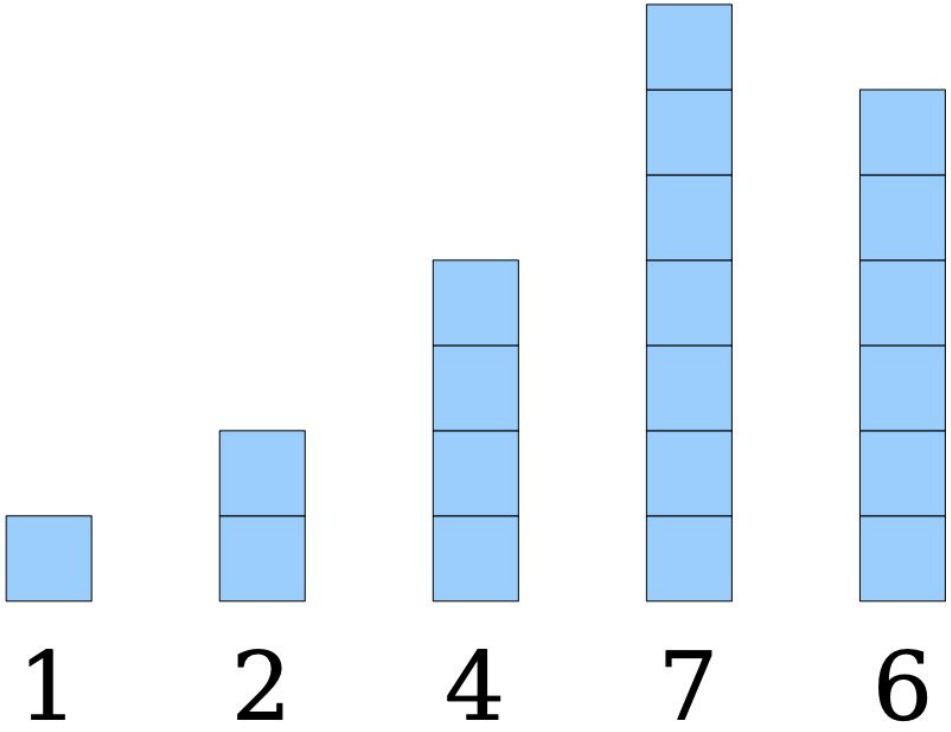
Our first sort: Selection sort



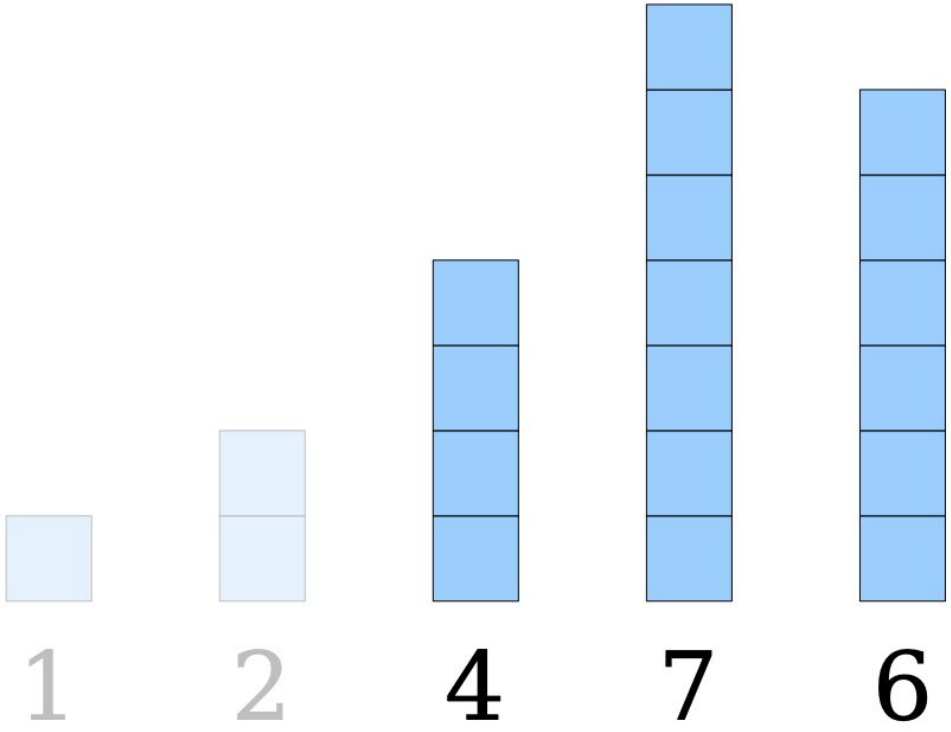
Our first sort: Selection sort



Our first sort: Selection sort

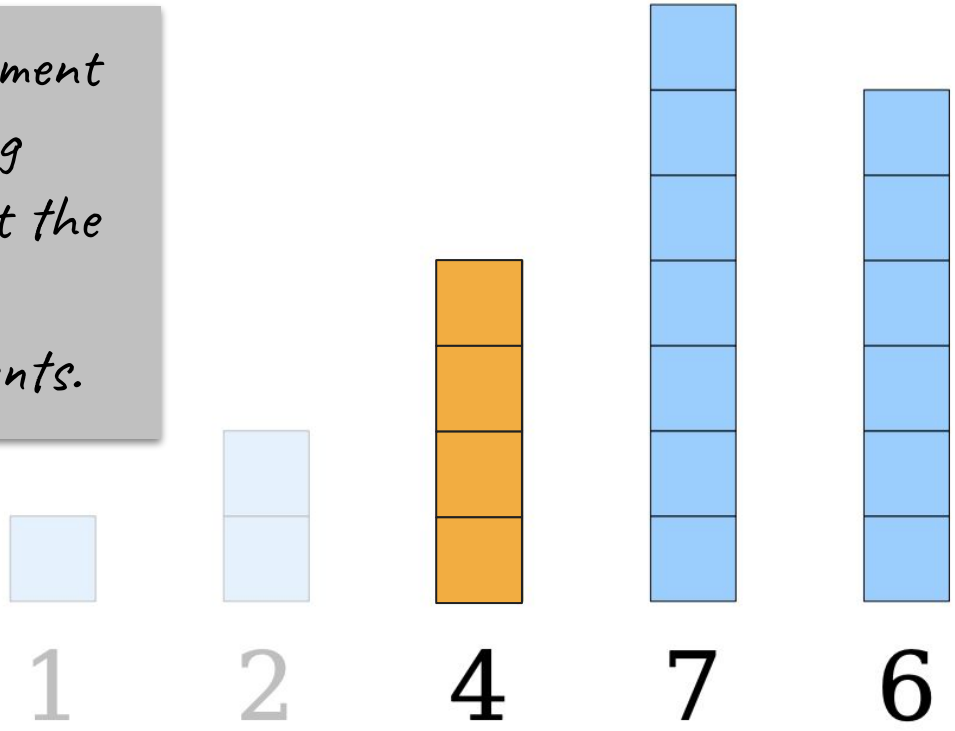


Our first sort: Selection sort

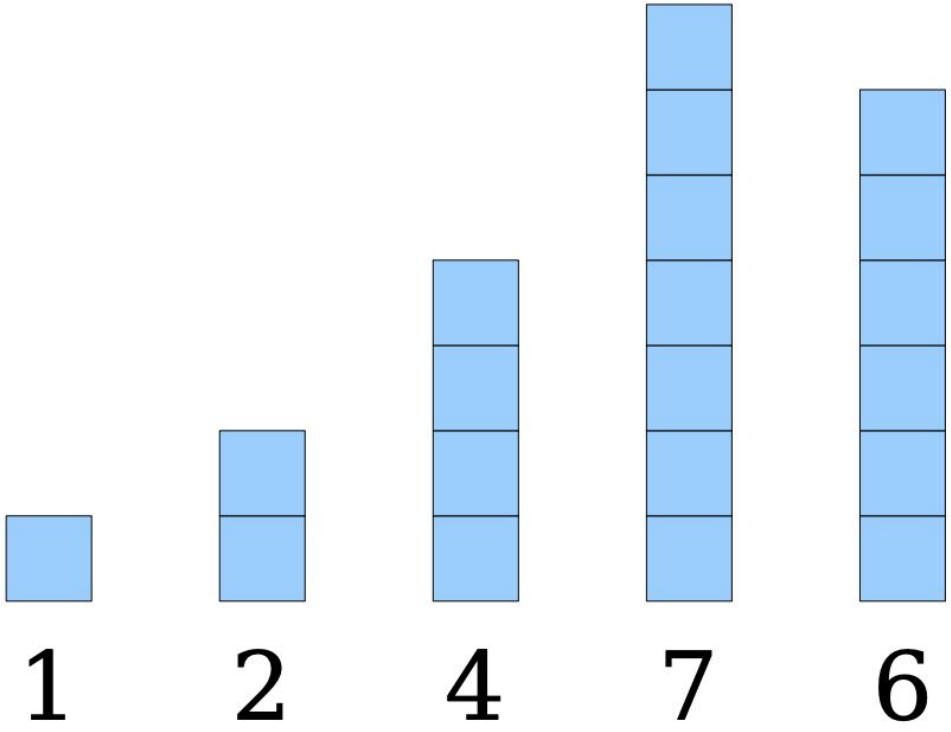


Our first sort: Selection sort

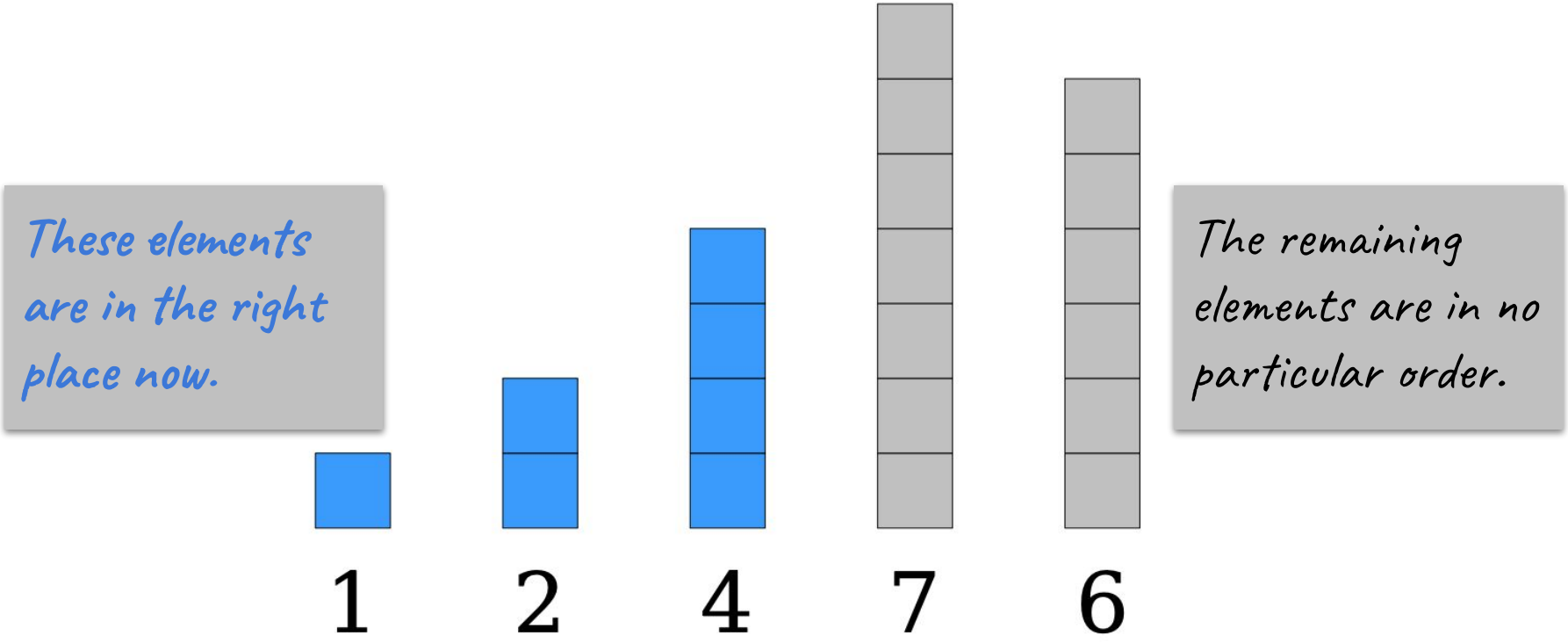
The smallest element of the remaining elements goes at the front of the remaining elements.



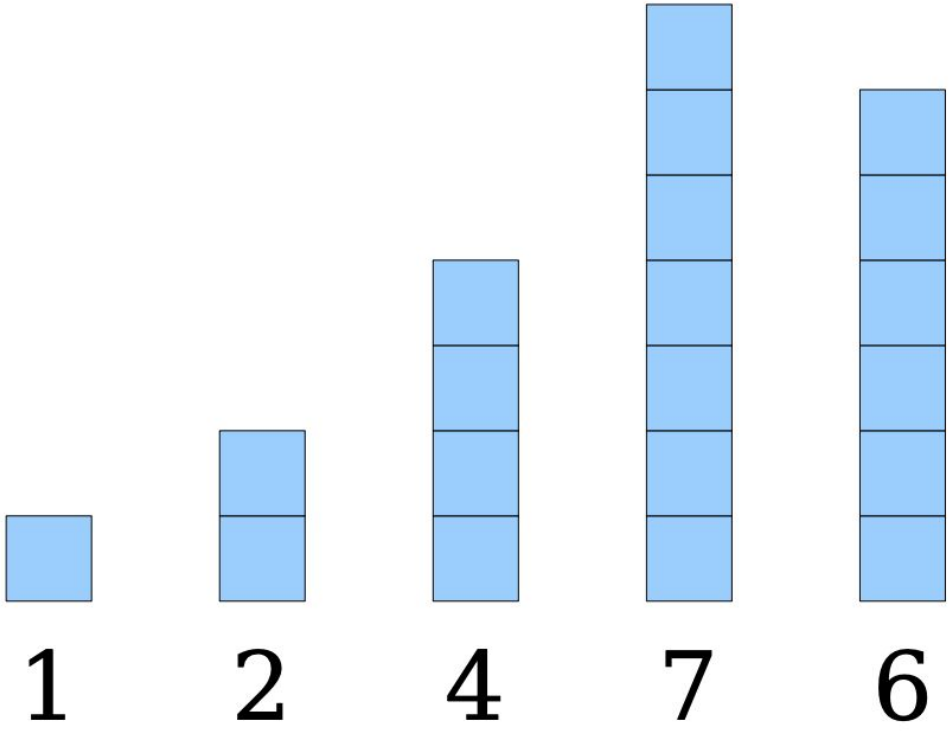
Our first sort: Selection sort



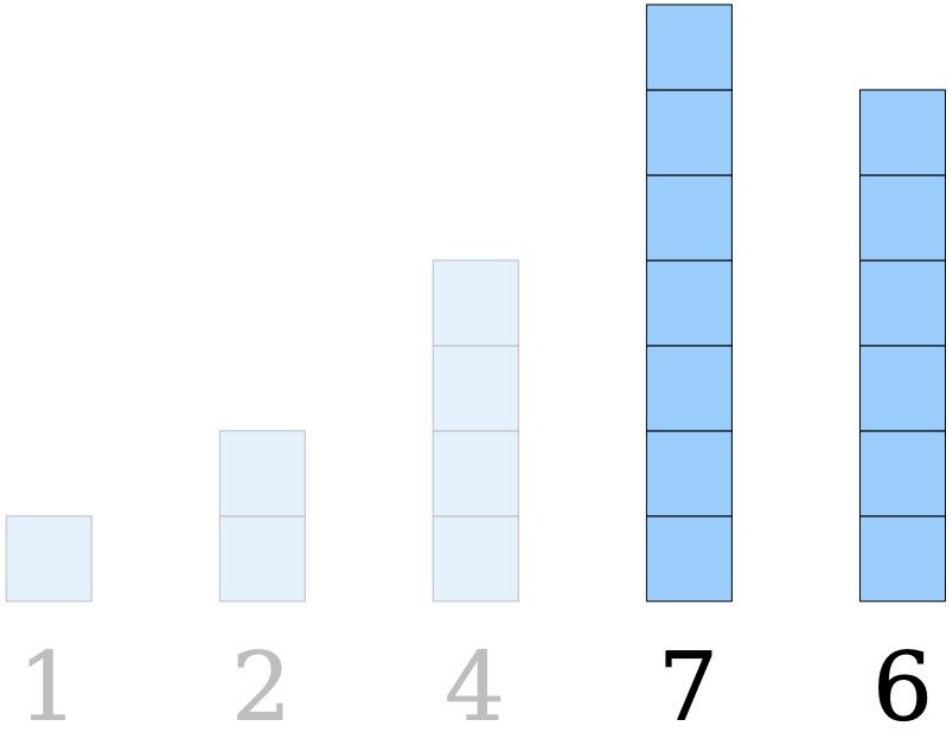
Our first sort: Selection sort



Our first sort: Selection sort

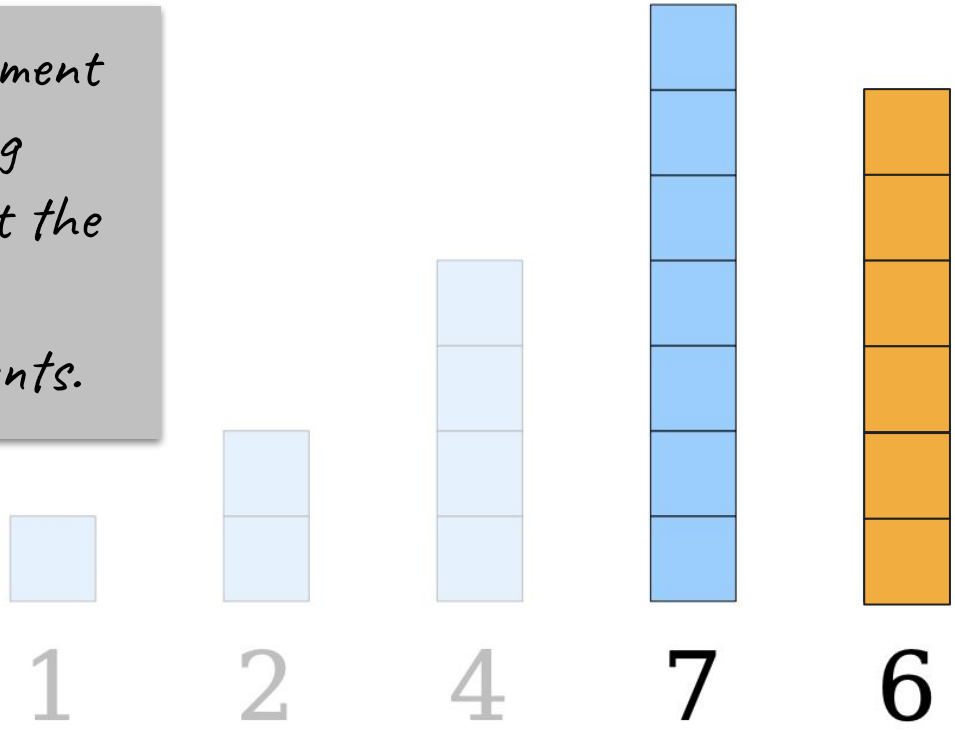


Our first sort: Selection sort

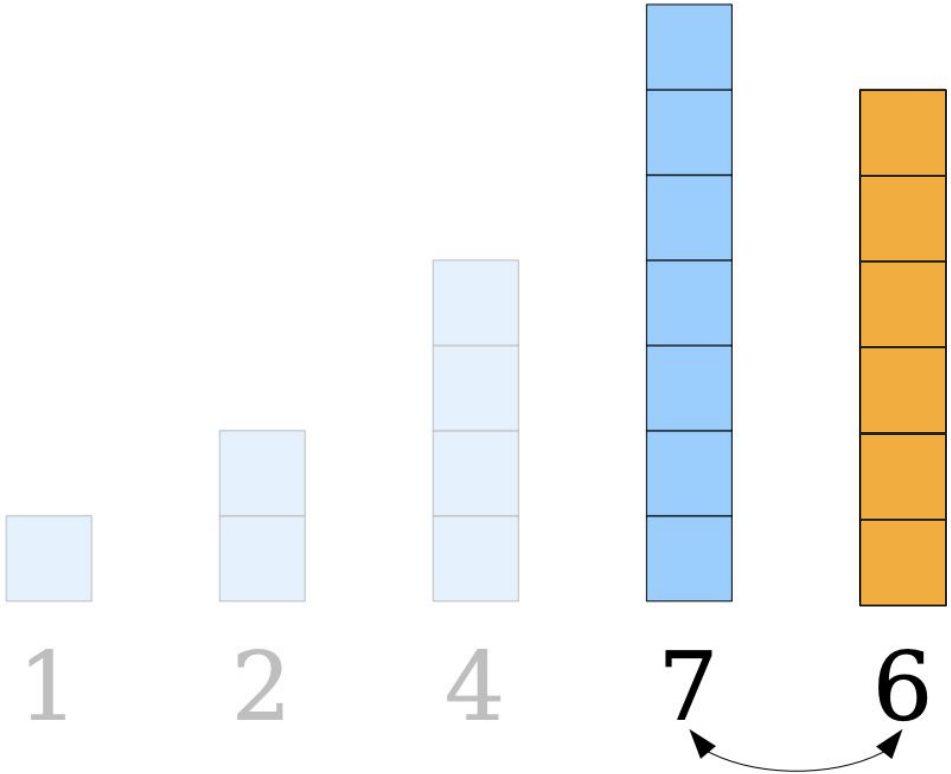


Our first sort: Selection sort

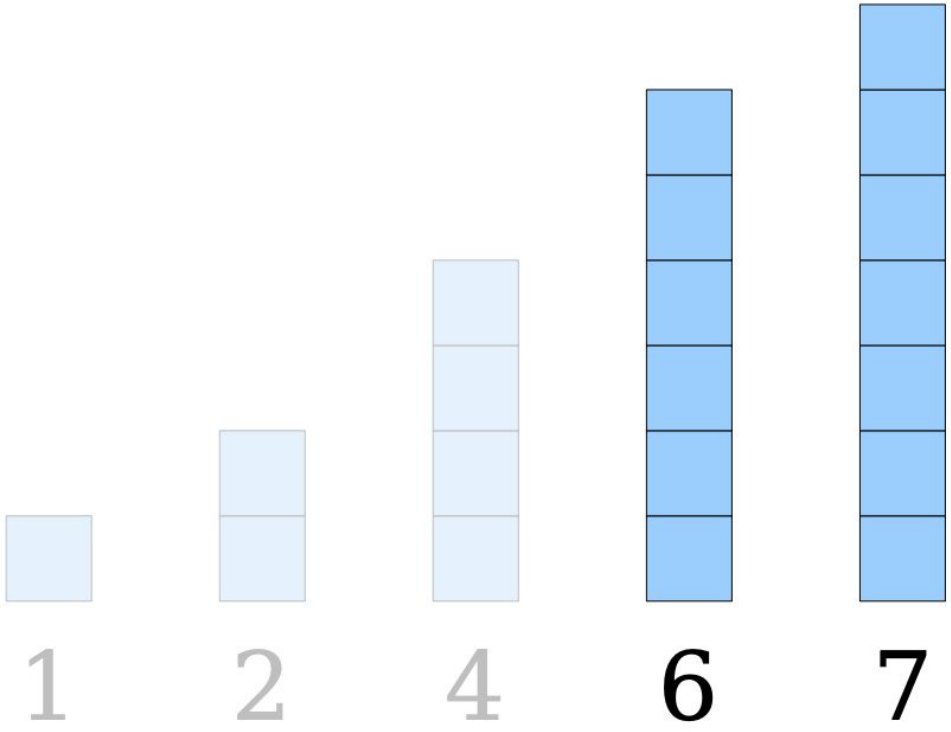
The smallest element of the remaining elements goes at the front of the remaining elements.



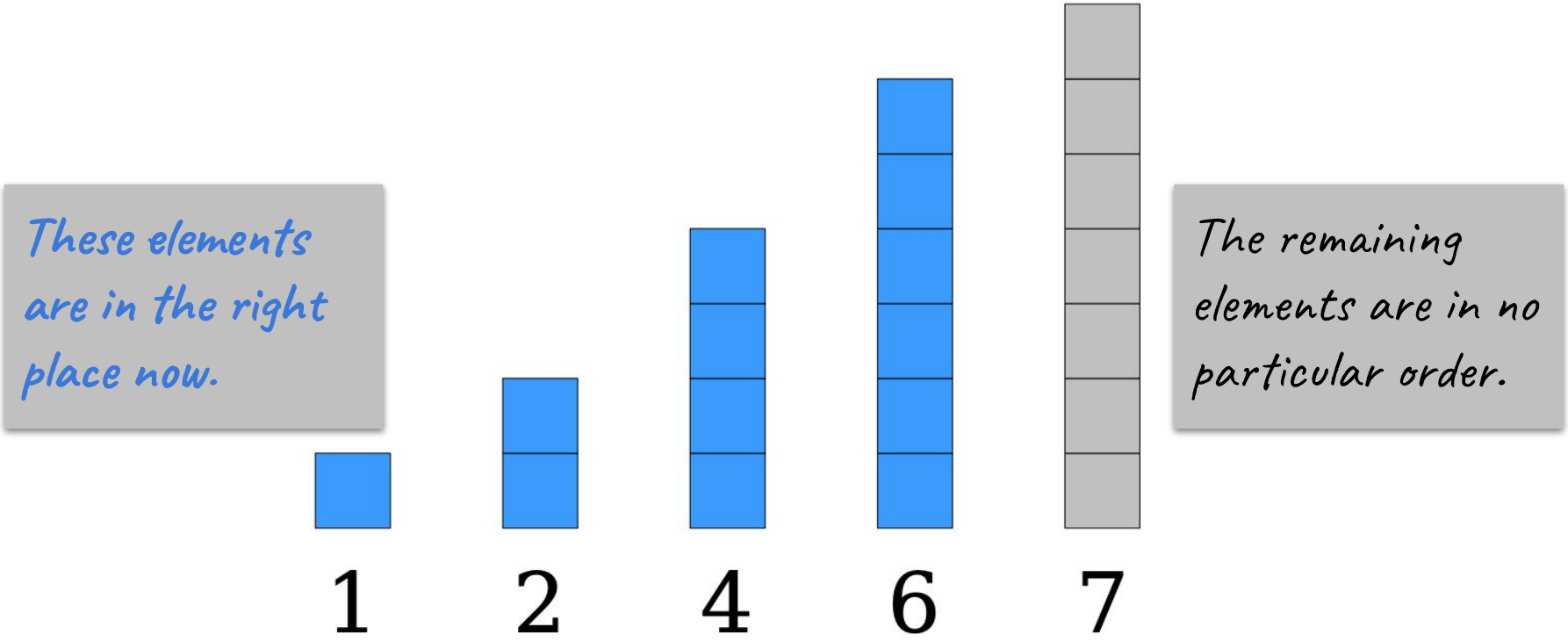
Our first sort: Selection sort



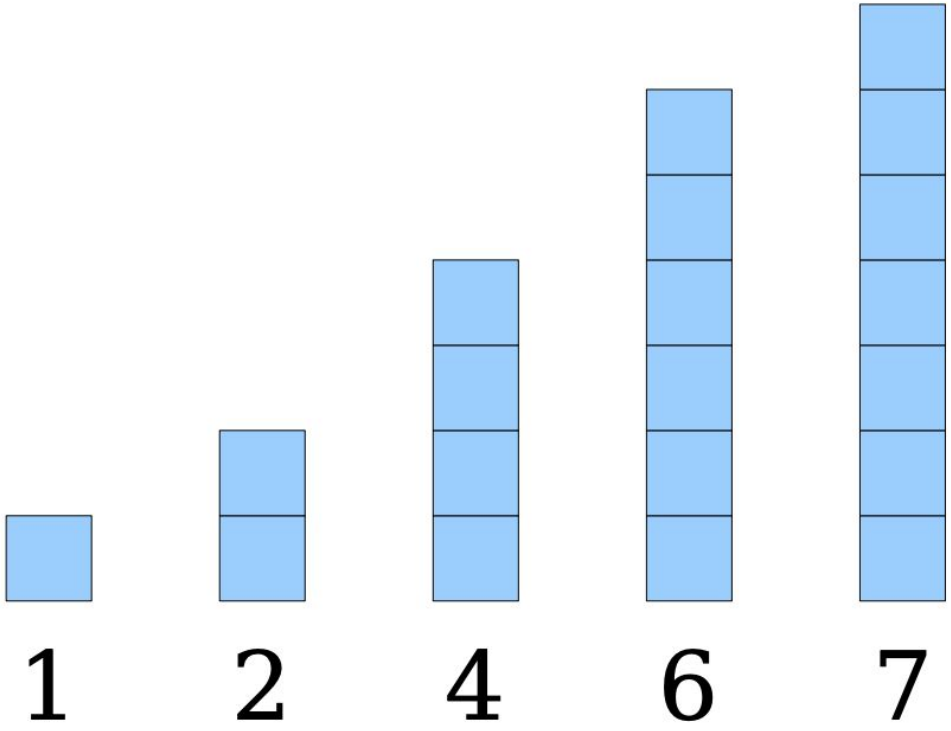
Our first sort: Selection sort



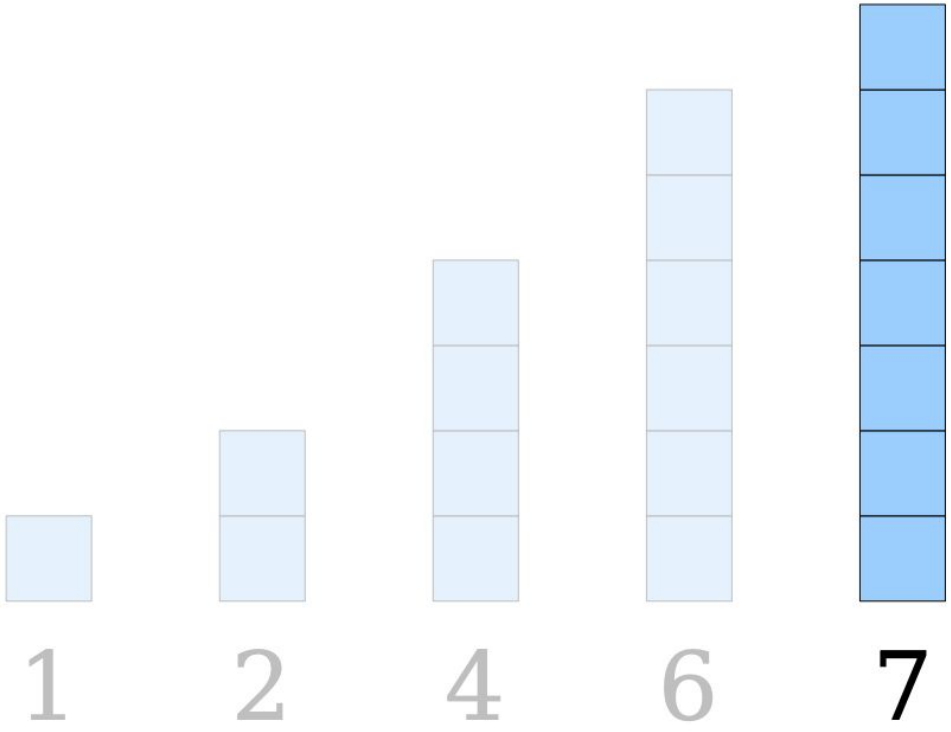
Our first sort: Selection sort



Our first sort: Selection sort

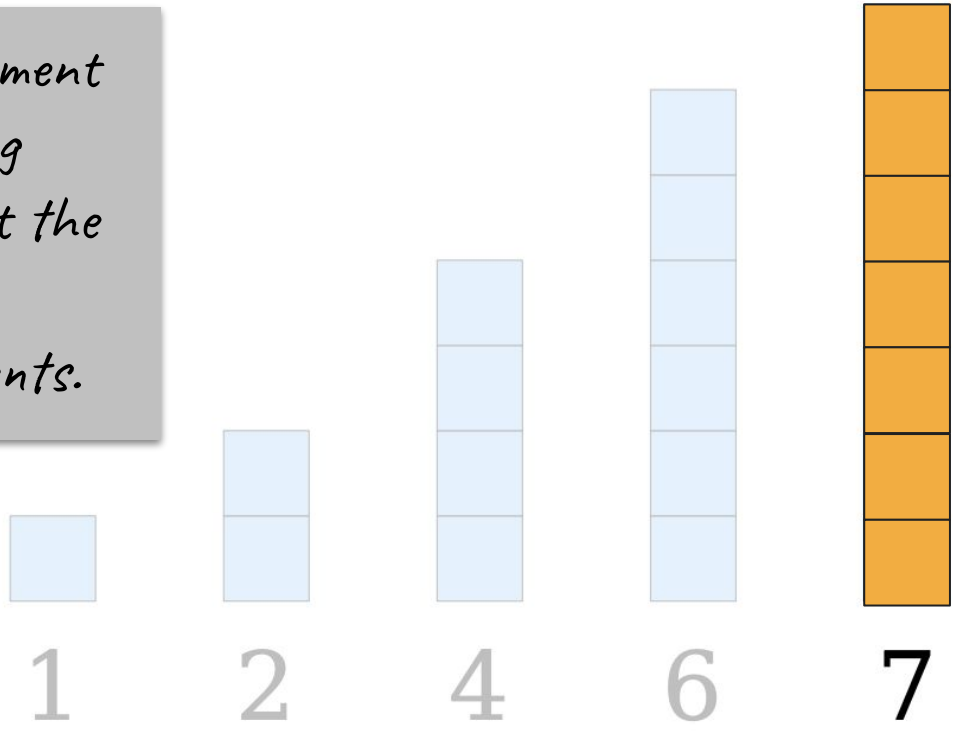


Our first sort: Selection sort

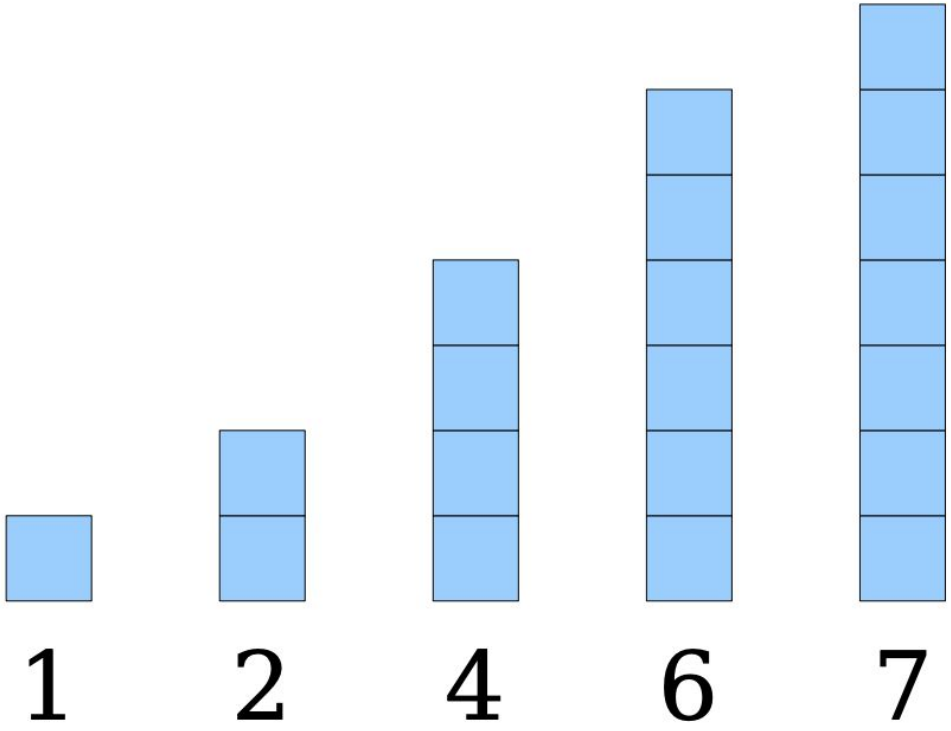


Our first sort: Selection sort

The smallest element of the remaining elements goes at the front of the remaining elements.

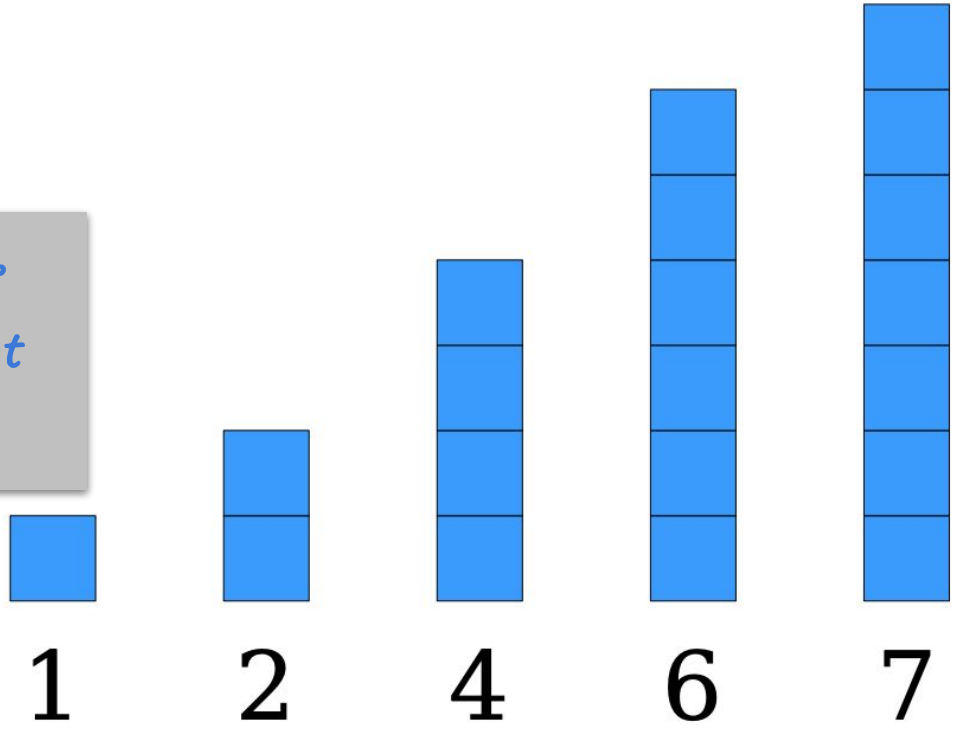


Our first sort: Selection sort



Our first sort: Selection sort

These elements are in the right place now.



Selection sort algorithm

- Find the smallest element and move it to the first position.
- Find the smallest element of what's left and move it to the second position.
- Find the smallest element of what's left and move it to the third position.
- Find the smallest element of what's left and move it to the fourth position.
- (etc.)

```
void selectionSort(Vector<int>& elems) {
    for (int index = 0; index < elems.size(); index++) {
        int smallestIndex = indexOfSmallest(elems, index);
        swap(elems[index], elems[smallestIndex]);
    }
}
/**
* Given a vector and a starting point, returns the index of the smallest
* element in that vector at or after the starting point
*/
int indexOfSmallest(const Vector<int>& elems, int startPoint) {
    int smallestIndex = startPoint;
    for (int i = startPoint + 1; i < elems.size(); i++) {
        if (elems[i] < elems[smallestIndex]) {
            smallestIndex = i;
        }
    }
    return smallestIndex;
}
```

Analyzing selection sort

- How much work do we do for selection sort?

Analyzing selection sort

- How much work do we do for selection sort?
  - To find the smallest value, we need to look at all `n` elements.

Analyzing selection sort

- How much work do we do for selection sort?
  - To find the smallest value, we need to look at all `n` elements.
  - To find the second-smallest value, we need to look at `n - 1` elements.

Analyzing selection sort

- How much work do we do for selection sort?
  - To find the smallest value, we need to look at all `n` elements.
  - To find the second-smallest value, we need to look at `n - 1` elements.
  - To find the third-smallest value, we need to look at `n - 2` elements.

Analyzing selection sort

- How much work do we do for selection sort?
  - To find the smallest value, we need to look at all **`n`** elements.
  - To find the second-smallest value, we need to look at **`n - 1`** elements.
  - To find the third-smallest value, we need to look at **`n - 2`** elements.
  - This process continues until we have found every last "smallest element" from the original collection.

Analyzing selection sort

- How much work do we do for selection sort?
  - To find the smallest value, we need to look at all `n` elements.
  - To find the second-smallest value, we need to look at `n - 1` elements.
  - To find the third-smallest value, we need to look at `n - 2` elements.
  - This process continues until we have found every last "smallest element" from the original collection.

- This, the total amount of work we have to do is is `n + (n - 1) + (n - 2) + ... + 1`

The complexity of selection sort

- There is a mathematical formula that tells us

```
n + (n-1) + ... + 2 + 1 = (n * (n+1)) / 2
```

The complexity of selection sort

- There is a mathematical formula that tells us

```
n + (n-1) + ... + 2 + 1 = (n * (n+1)) / 2
```

- Thus, the overall complexity of selection sort can be simplified as follows:
  - Total work = `O((n * (n+1)) / 2)`

The complexity of selection sort

- There is a mathematical formula that tells us

```
n + (n-1) + ... + 2 + 1 = (n * (n+1)) / 2
```

- Thus, the overall complexity of selection sort can be simplified as follows:
  - Total work = `O((n * (n+1)) / 2)`

`= O(n * (n+1))` ← *Big-O ignores constant factors*

The complexity of selection sort

- There is a mathematical formula that tells us

$$n + (n-1) + \ldots + 2 + 1 = (n * (n+1)) / 2$$

- Thus, the overall complexity of selection sort can be simplified as follows:
  - Total work $= O((n * (n+1)) / 2)$

$= O(n * (n+1))$ ← *Big-O ignores constant factors*

$= O(n^2 + n)$

The complexity of selection sort

- There is a mathematical formula that tells us

$$n + (n-1) + \ldots + 2 + 1 = (n * (n+1)) / 2$$

- Thus, the overall complexity of selection sort can be simplified as follows:
  - Total work $= O((n * (n+1)) / 2)$

$= O(n * (n+1))$ ← Big-O ignores constant factors

$= O(n^2 + n)$

$= O(n^2)$ ← Big-O ignores low-order terms

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.

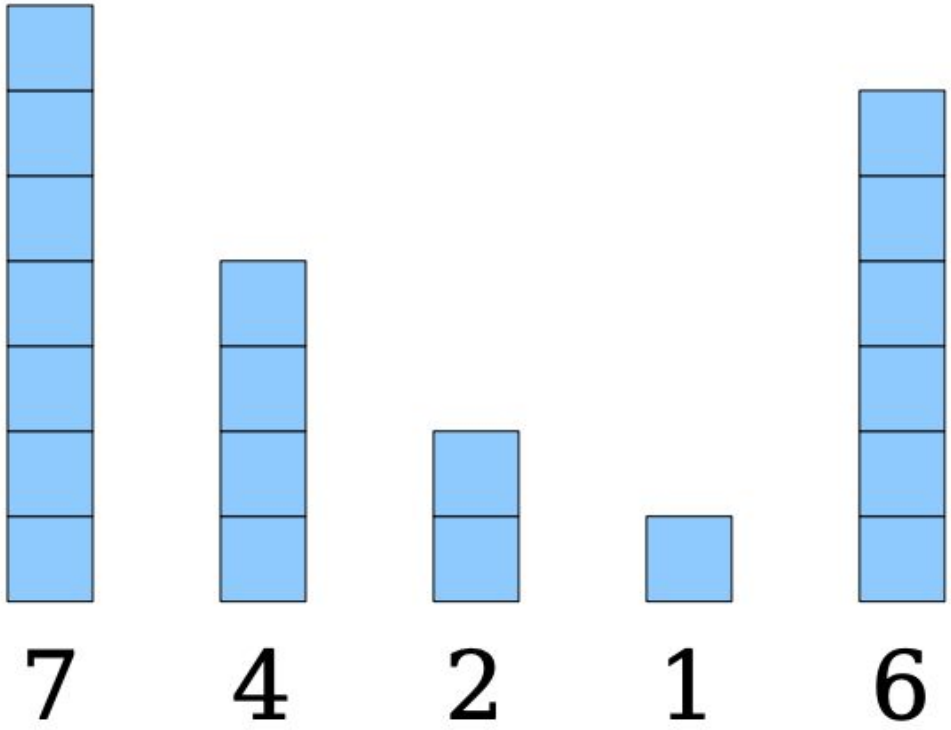
- Selection sort is an $O(n^2)$ algorithm.

- Can we do better?
  - Yes!

Insertion Sort

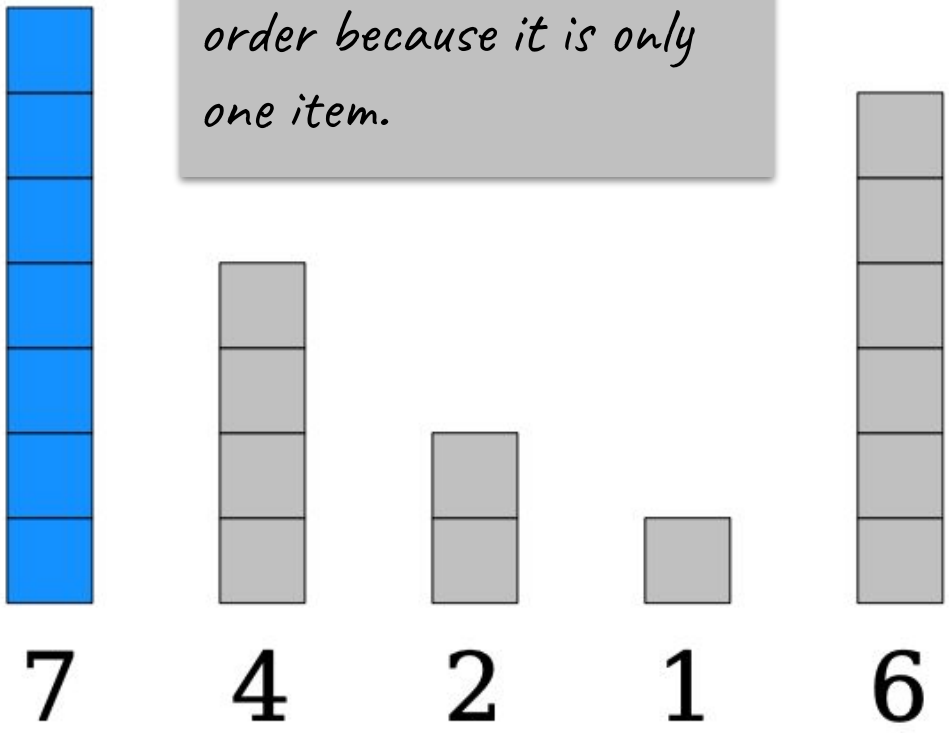
(Bonus Content, not covered in live lecture)

Insertion sort

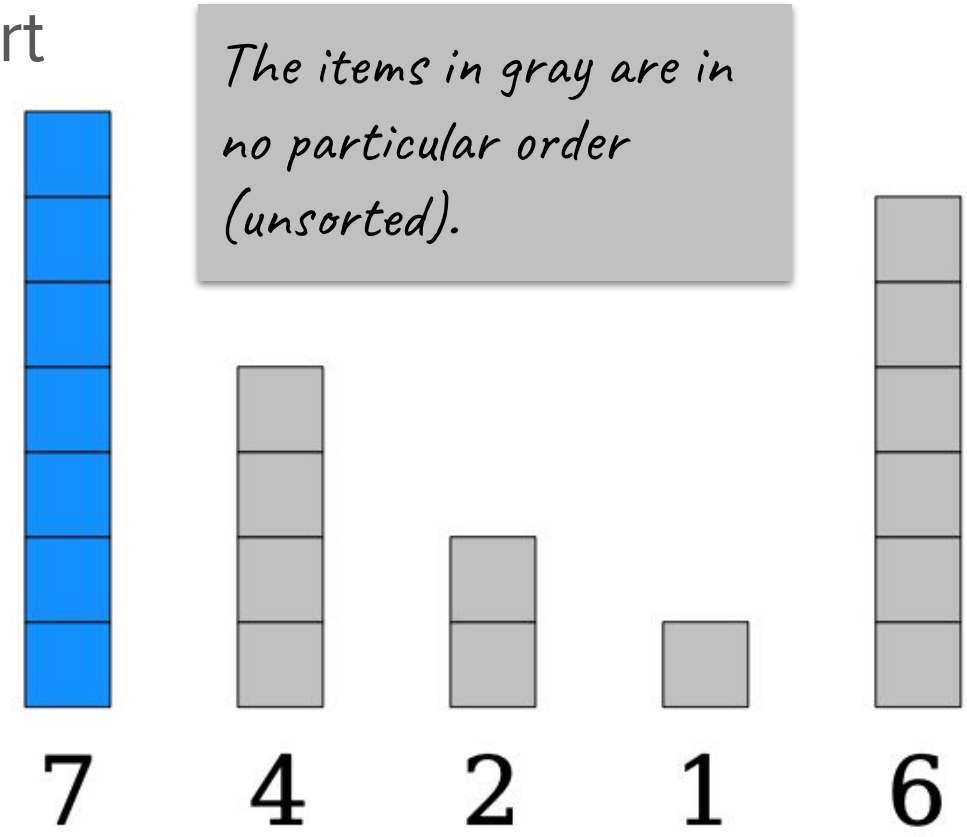


Insertion sort

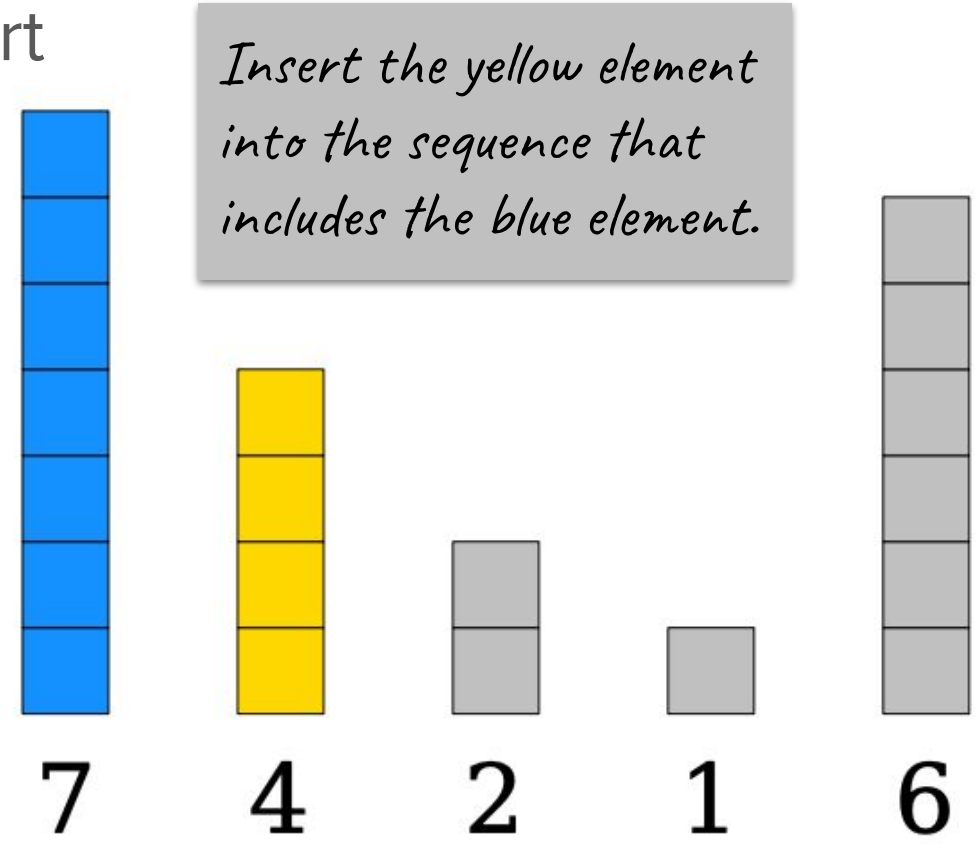
Considered alone, the blue item is trivially in sorted order because it is only one item.



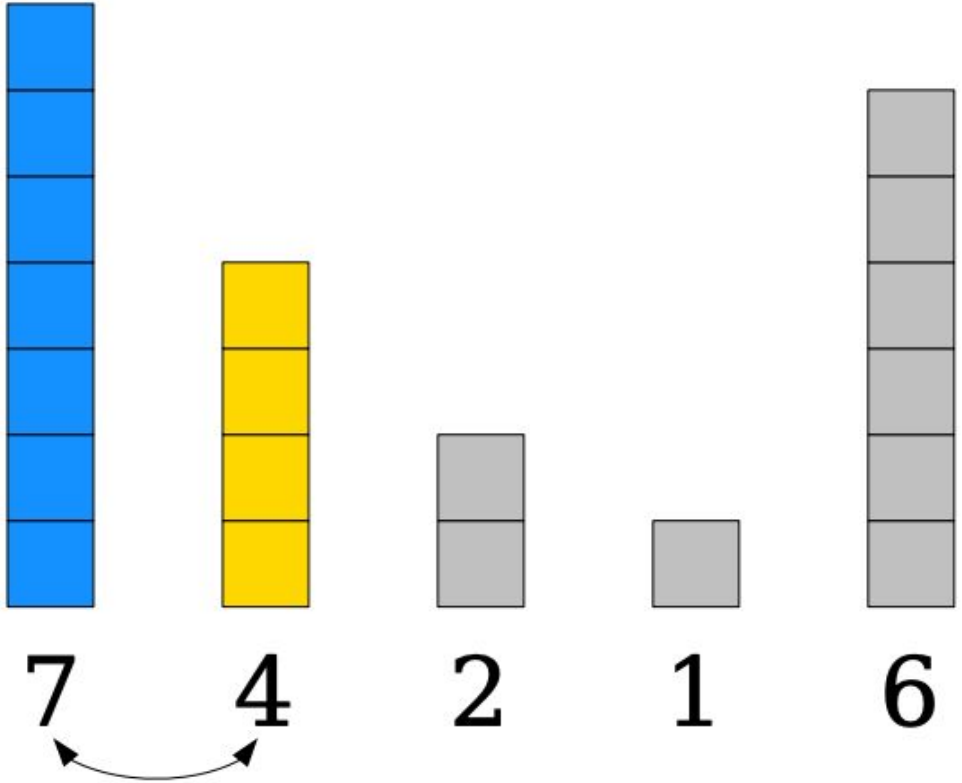
Insertion sort



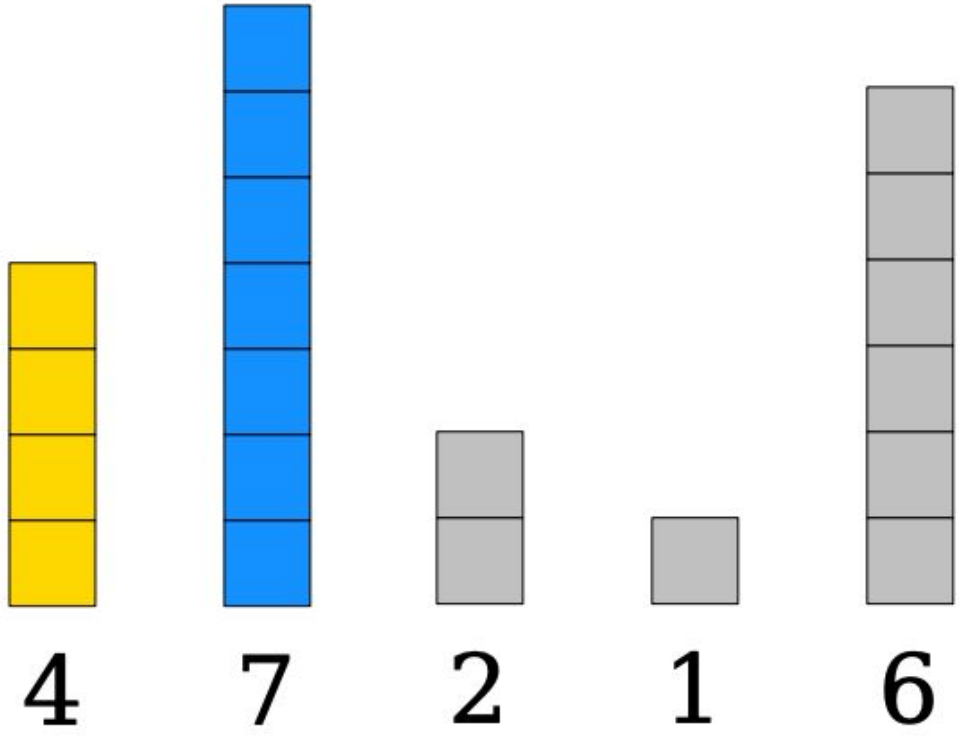
Insertion sort



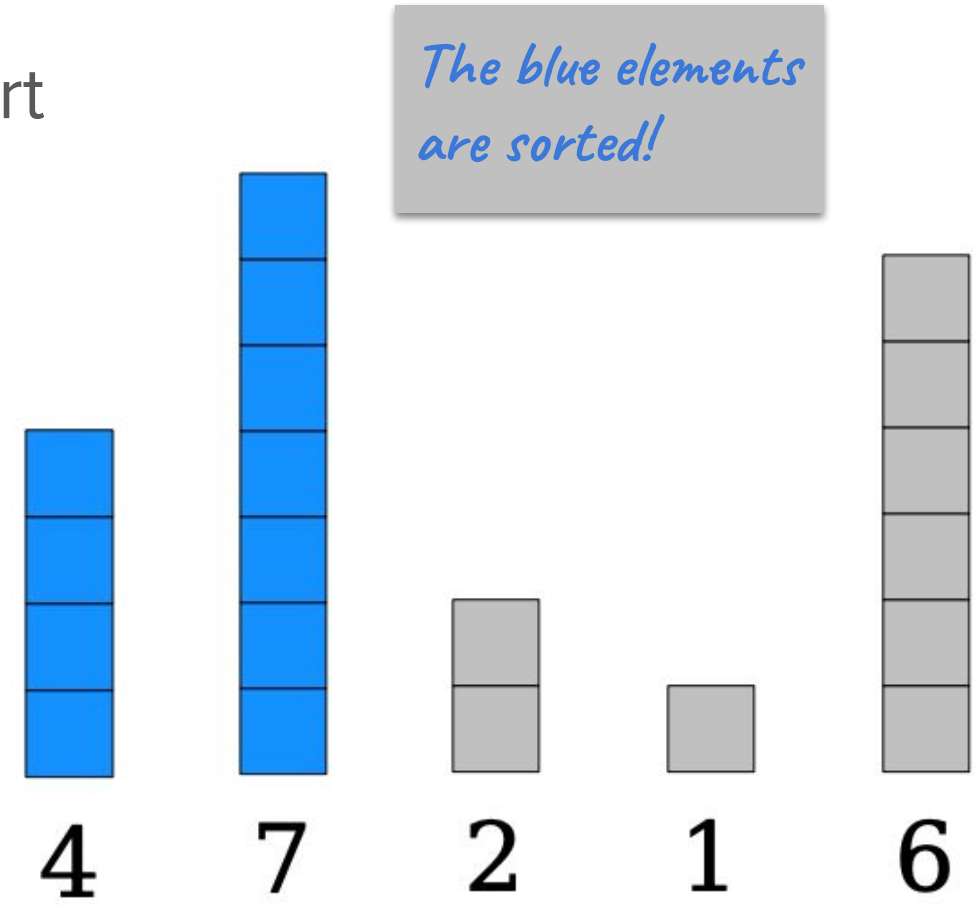
Insertion sort



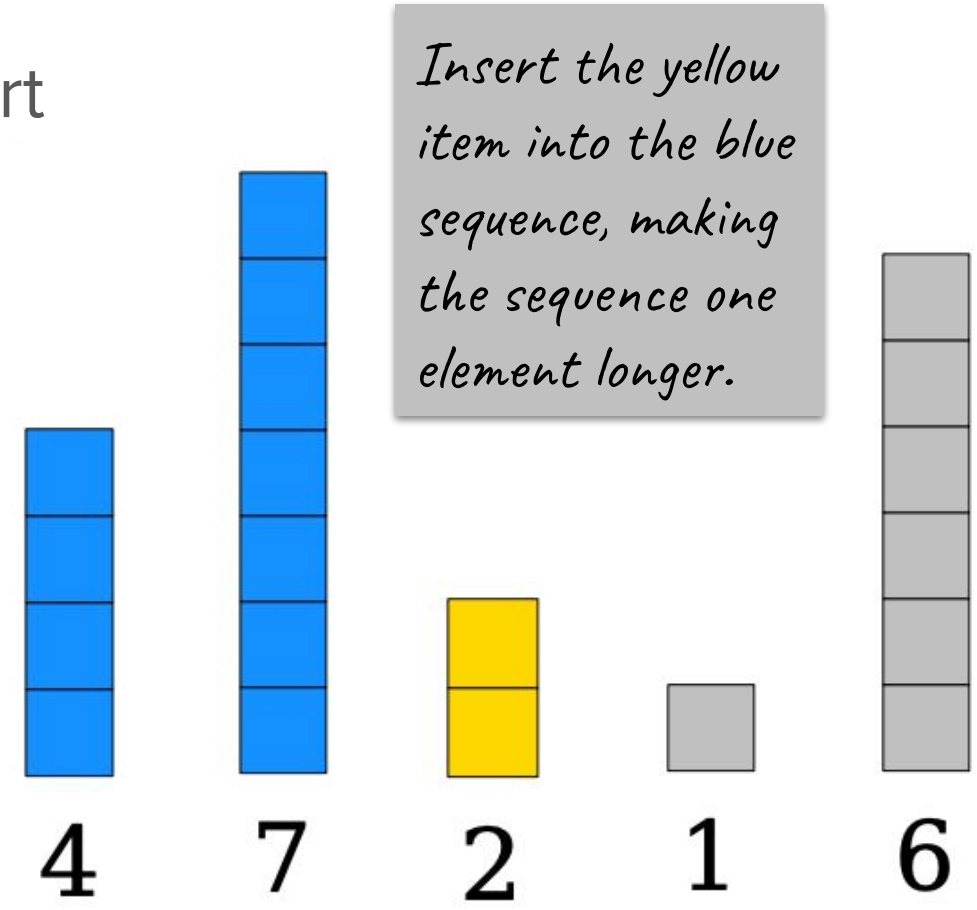
Insertion sort



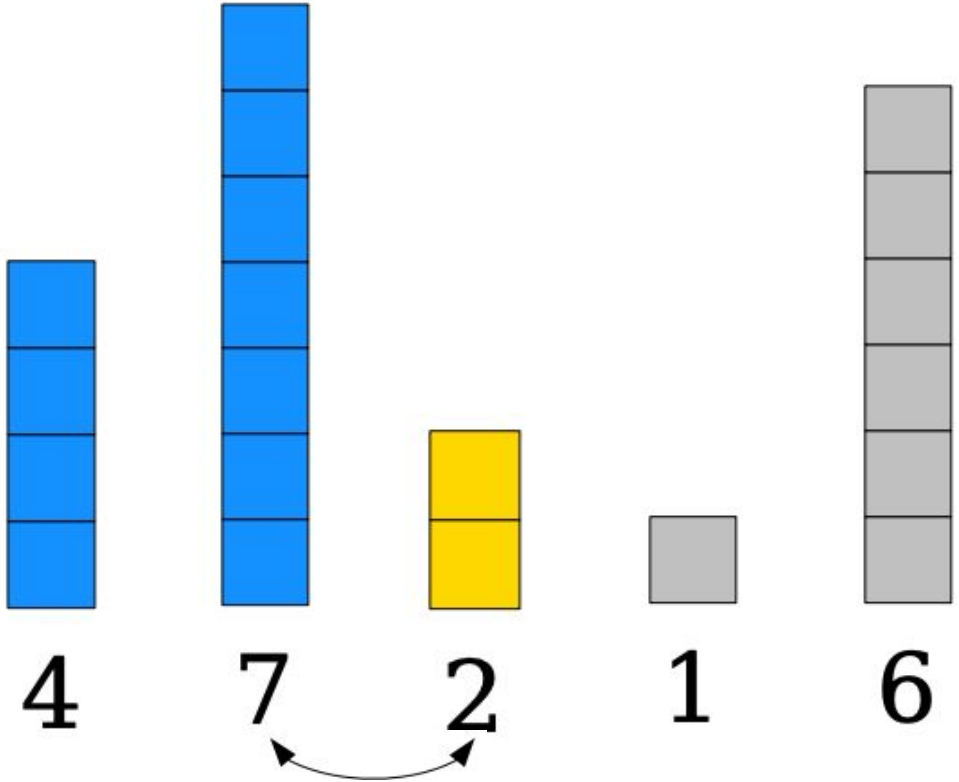
Insertion sort



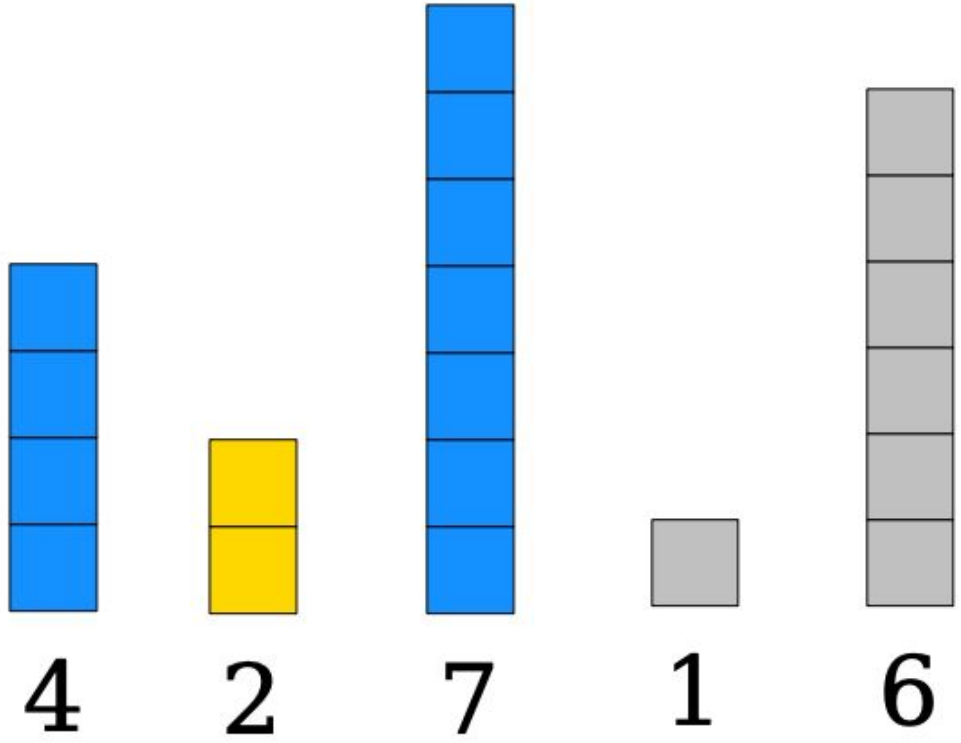
Insertion sort



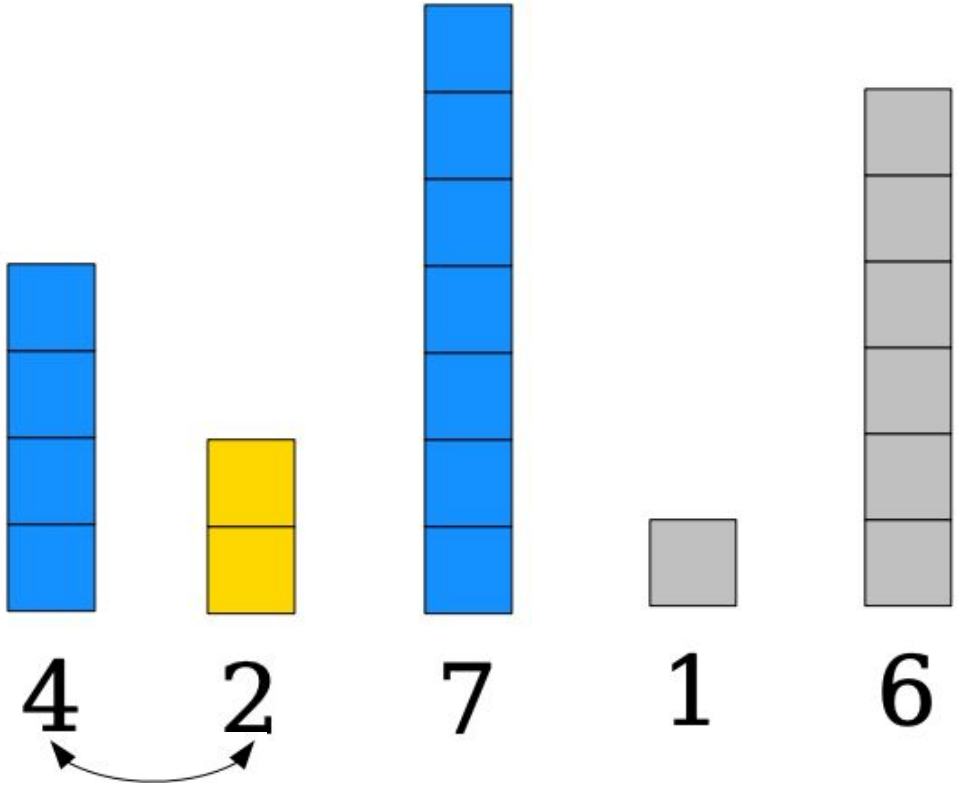
Insertion sort



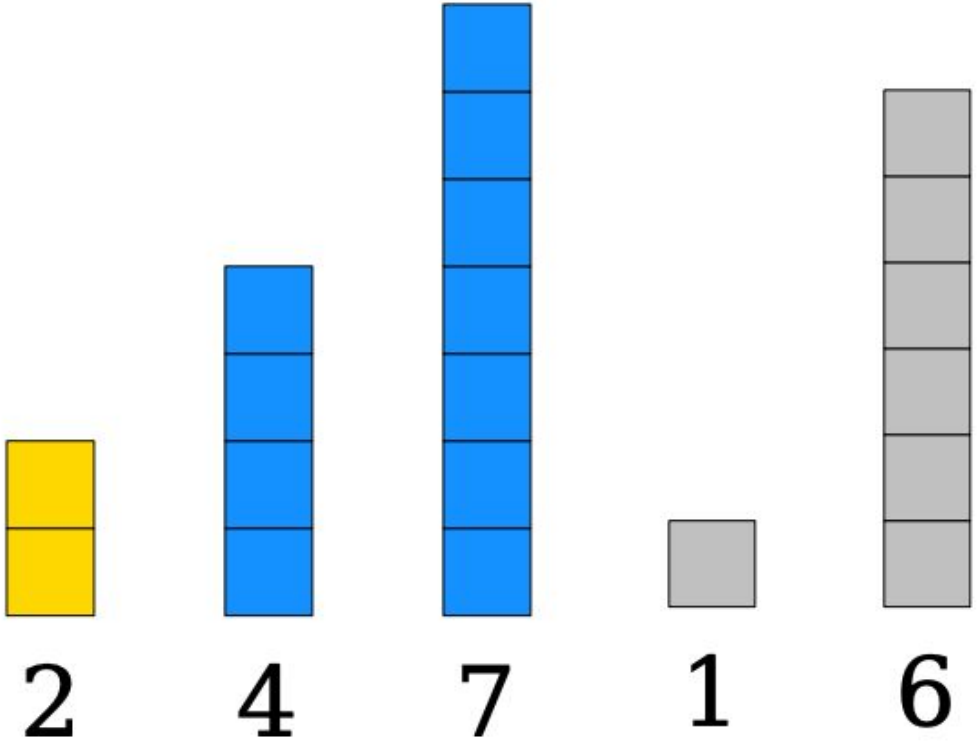
Insertion sort



Insertion sort

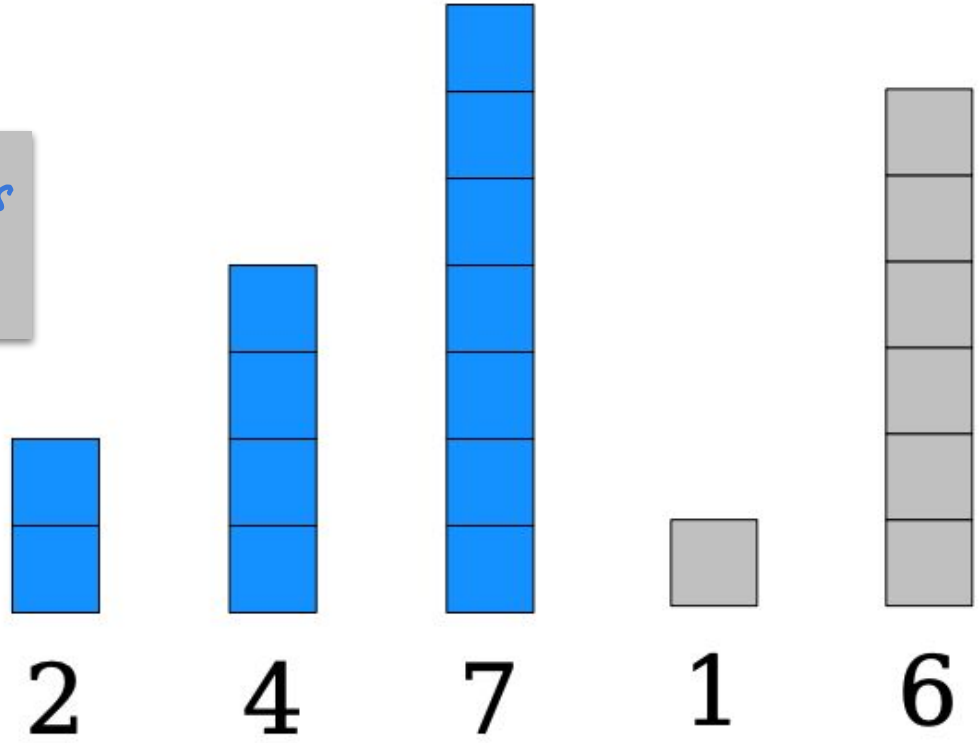


Insertion sort



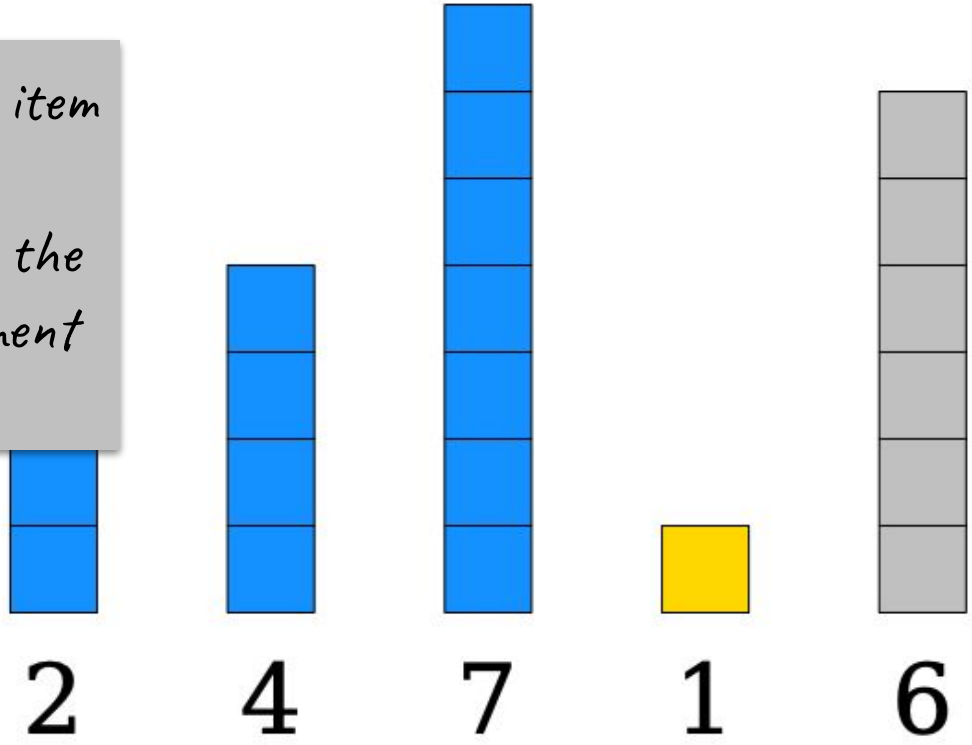
Insertion sort

The blue elements are sorted!

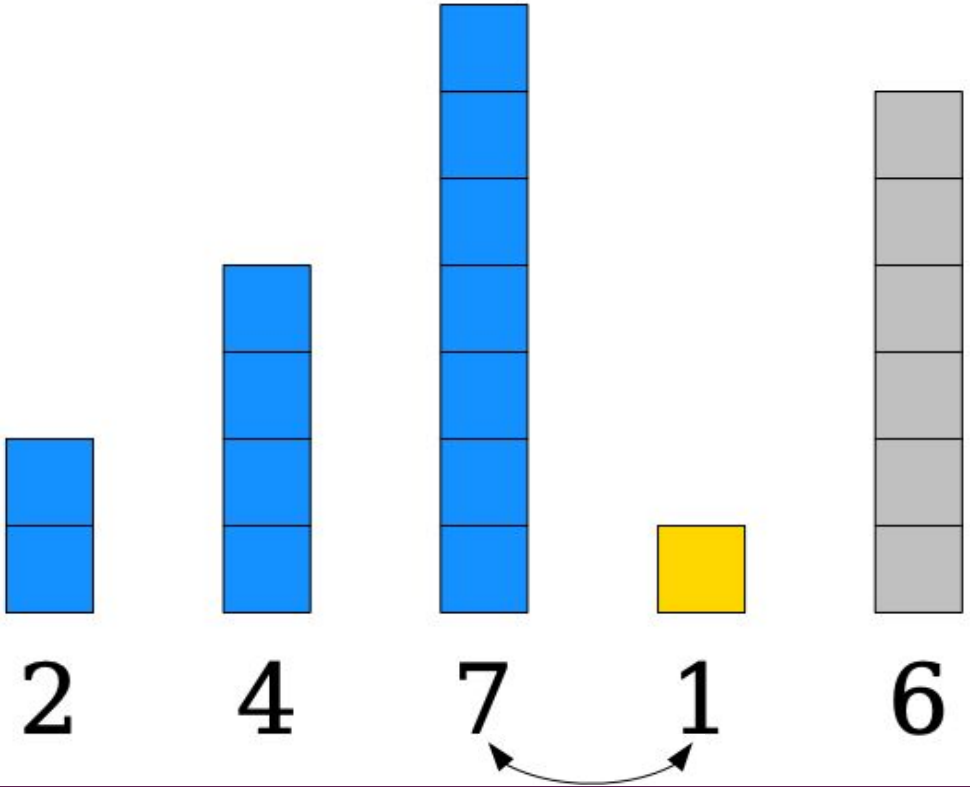


Insertion sort

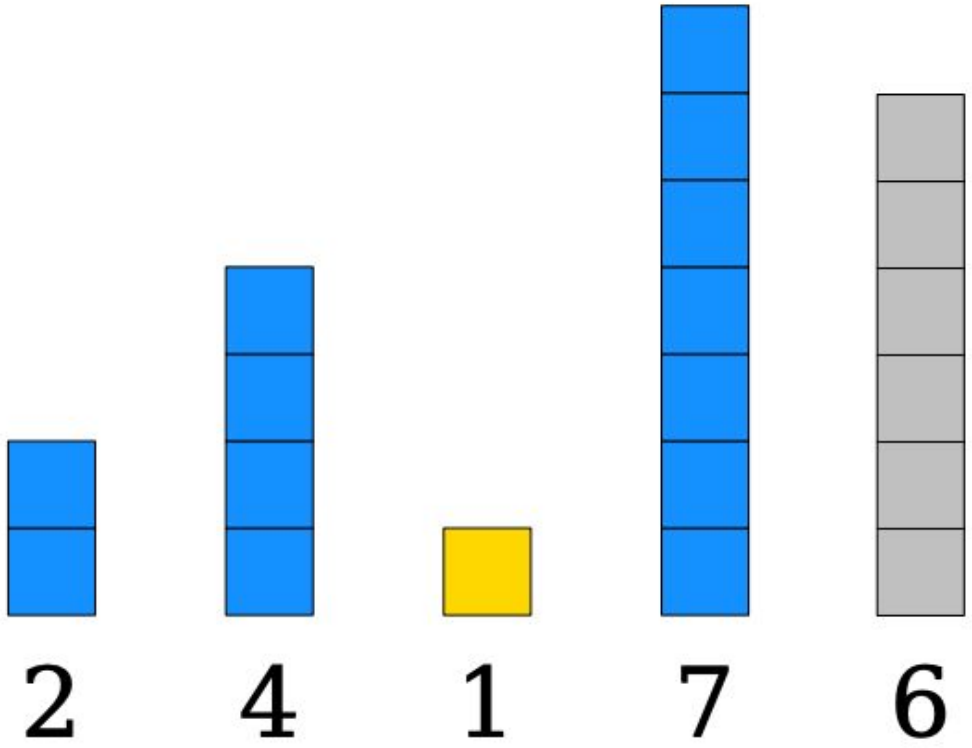
Insert the yellow item into the blue sequence, making the sequence one element longer.



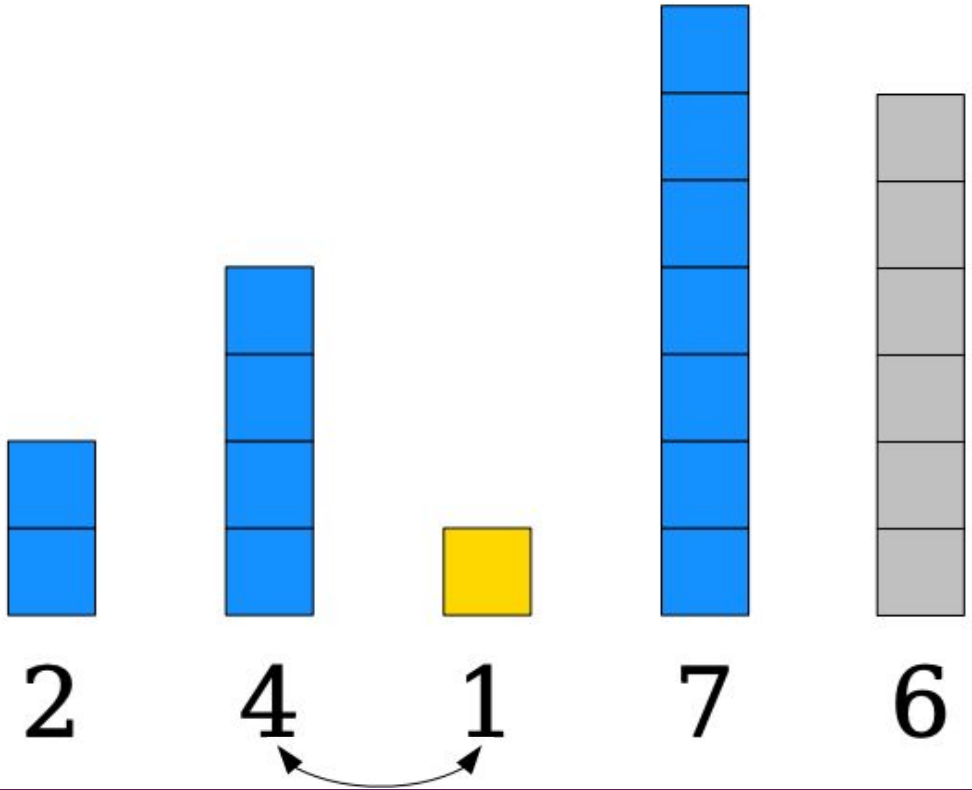
Insertion sort



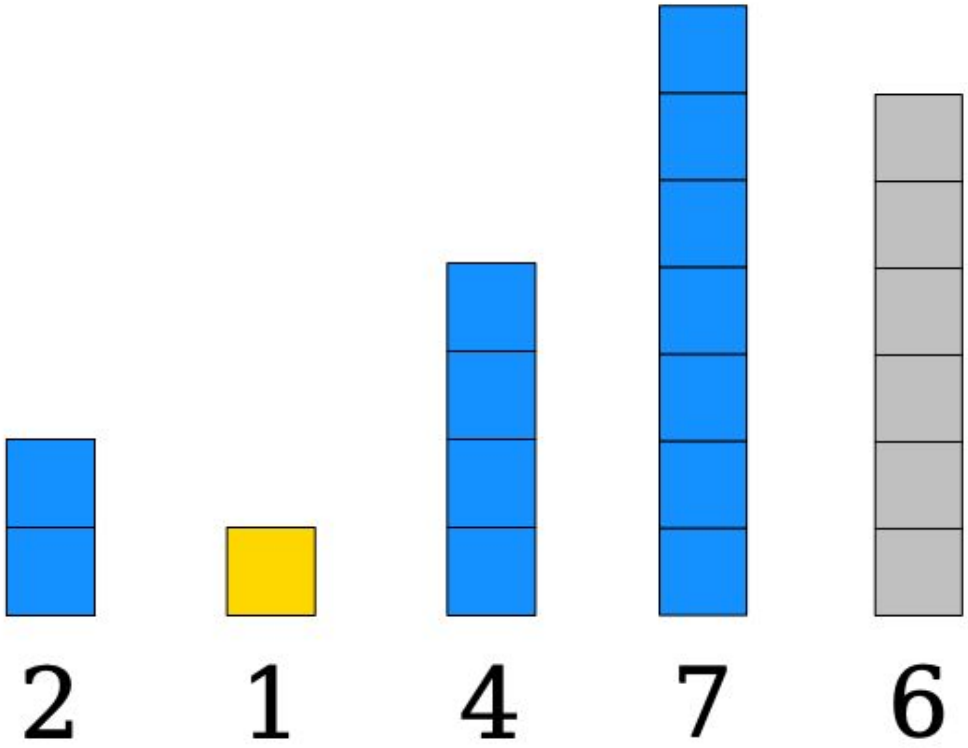
Insertion sort



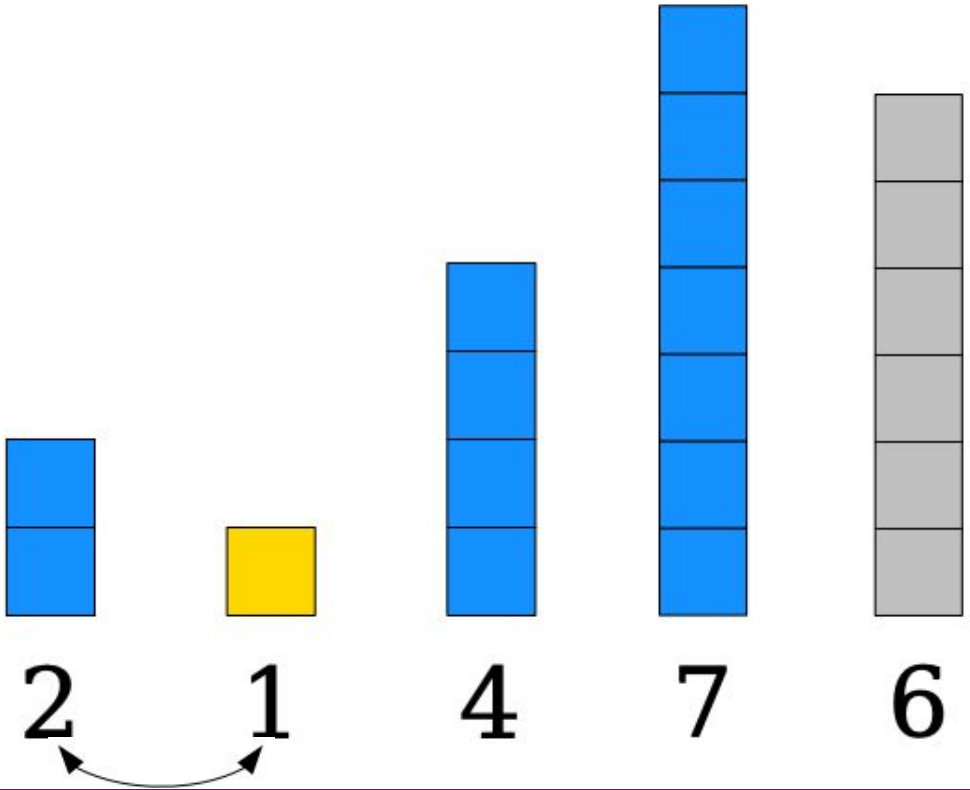
Insertion sort



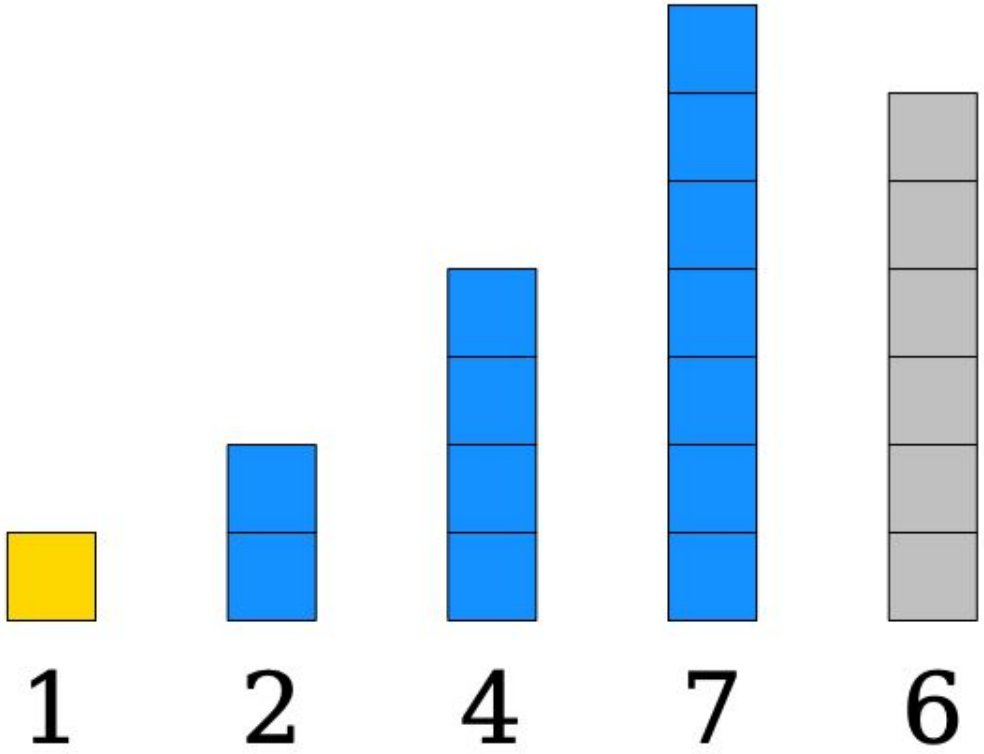
Insertion sort



Insertion sort

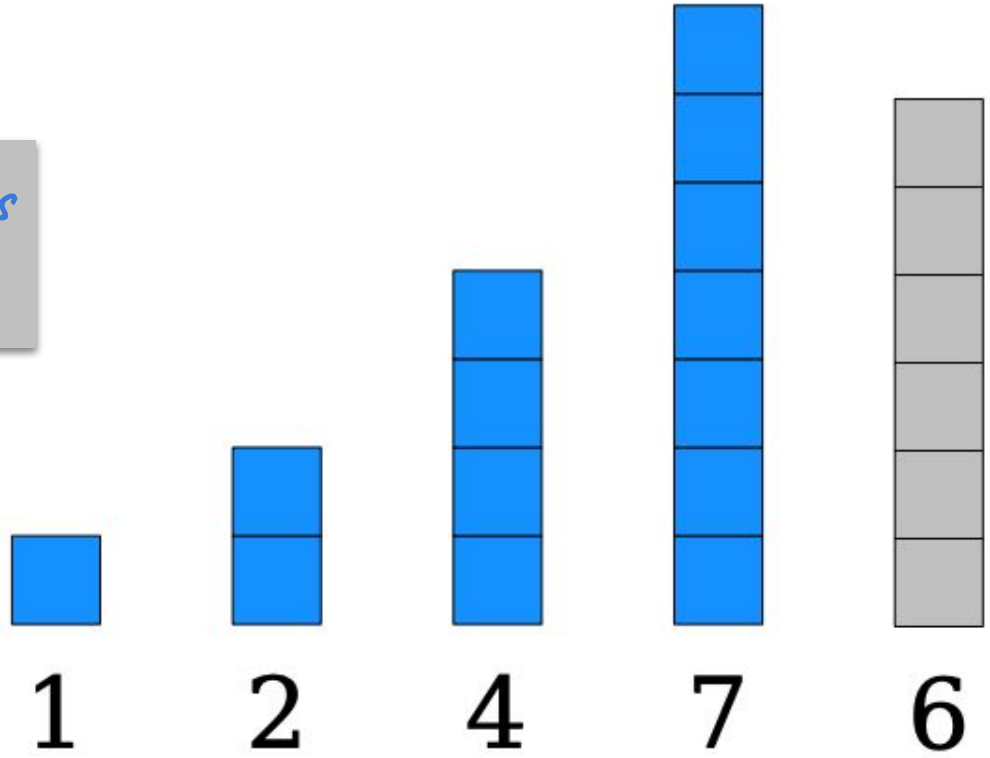


Insertion sort



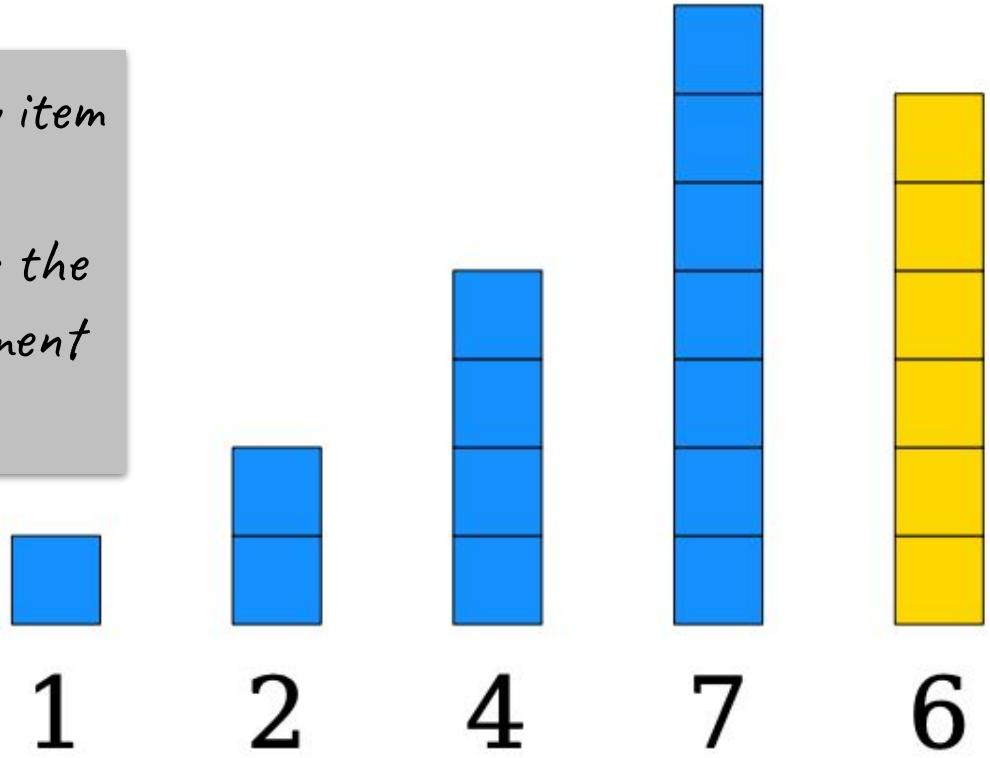
Insertion sort

The blue elements are sorted!

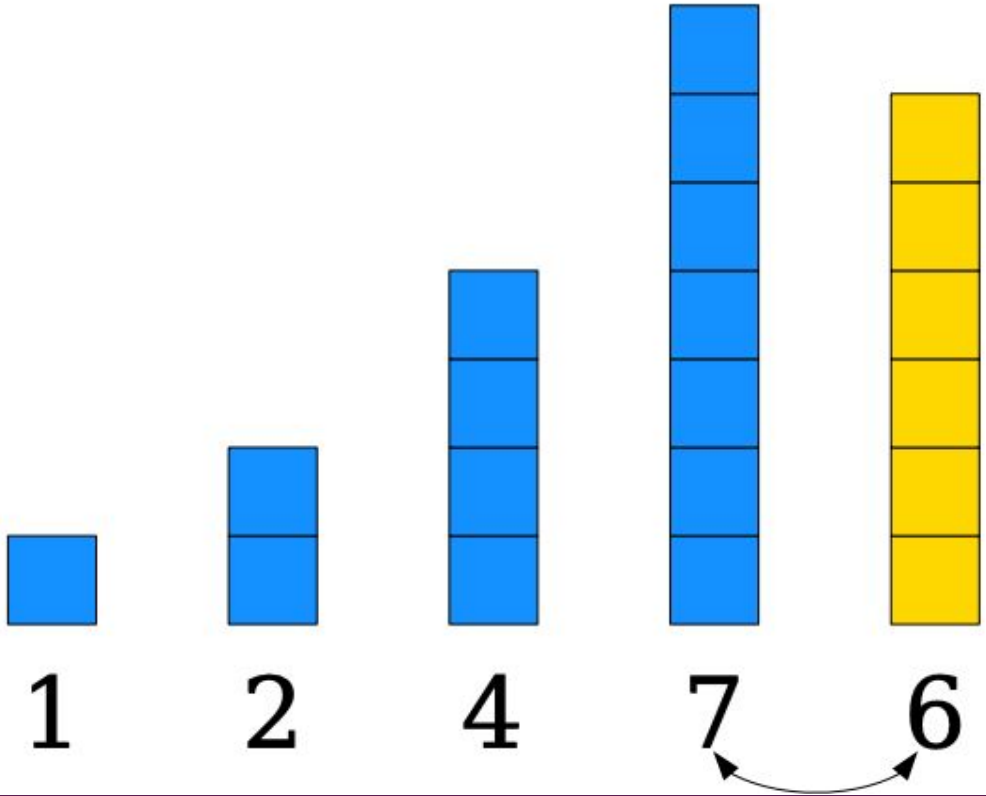


Insertion sort

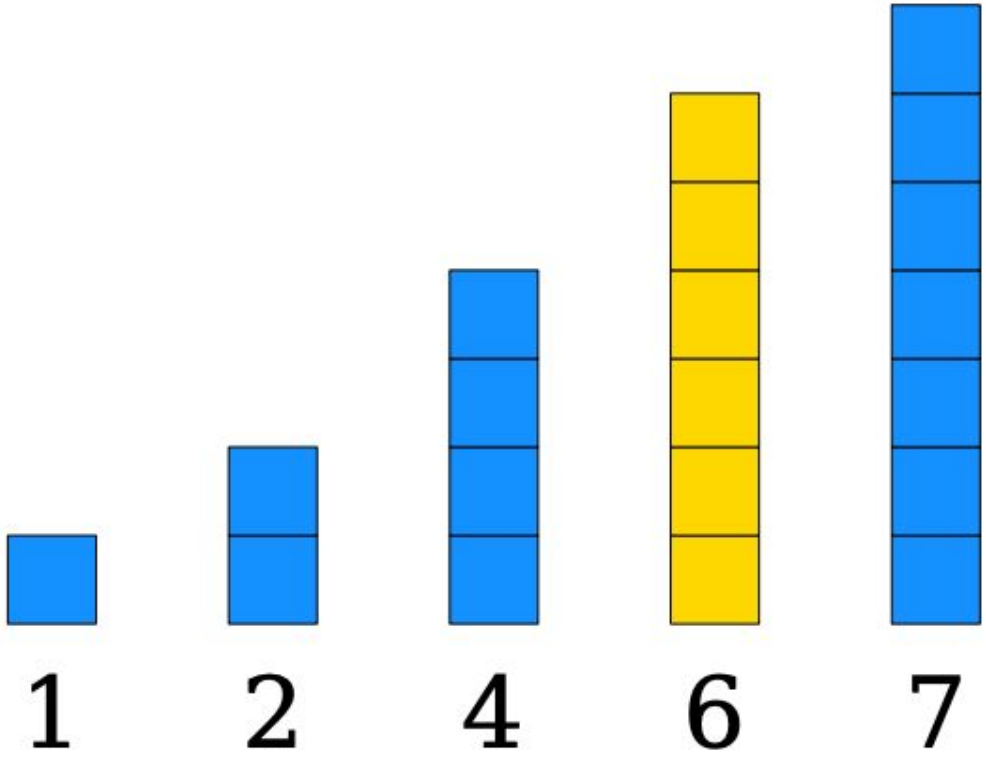
Insert the yellow item into the blue sequence, making the sequence one element longer.



Insertion sort

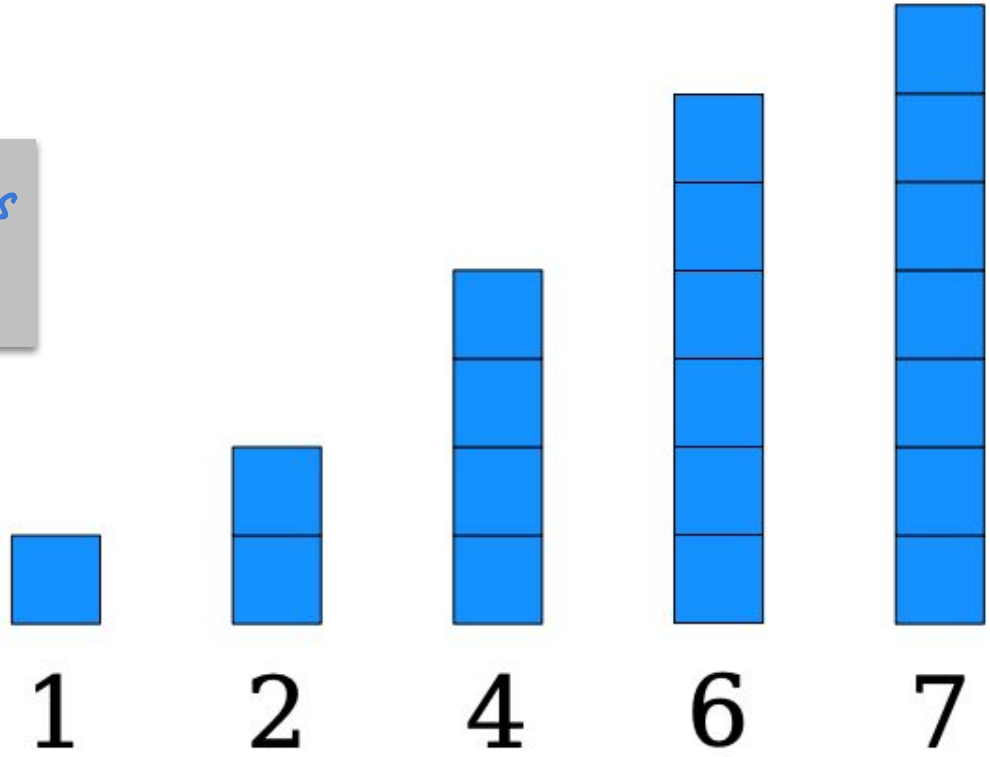


Insertion sort



Insertion sort

The blue elements are sorted!



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.

Insertion sort code

```
void insertionSort(Vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        /* Scan backwards until either (1) the preceding
         * element is no bigger than us or (2) there is
         * no preceding element. */
        for (int j = i - 1; j >= 0; j--) {
            if (v[j] <= v[j + 1]) break;
            /* Swap this element back one step. */
            swap(v[j], v[j + 1]);
        }
    }
}
```

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 CS109 if you're interested in learning more!

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:

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

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:

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

- Poll: Approximately how many seconds will it take to run 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!

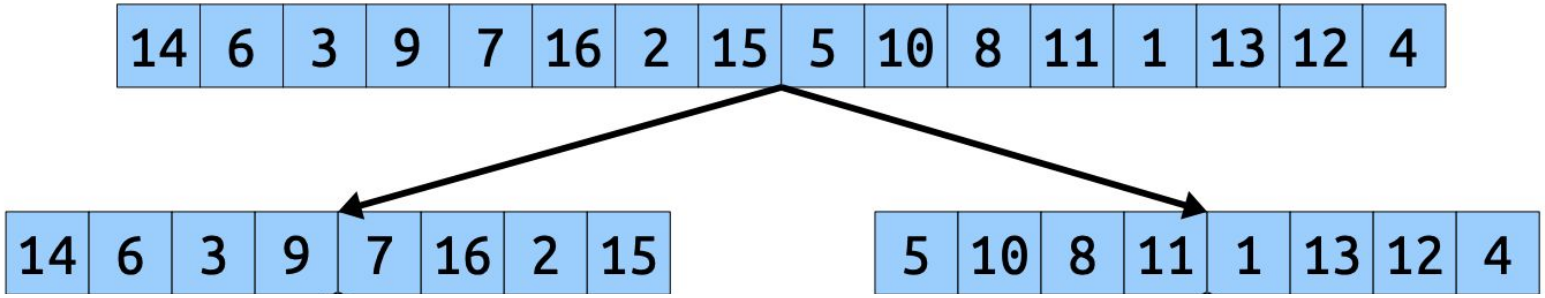
Merge Sort

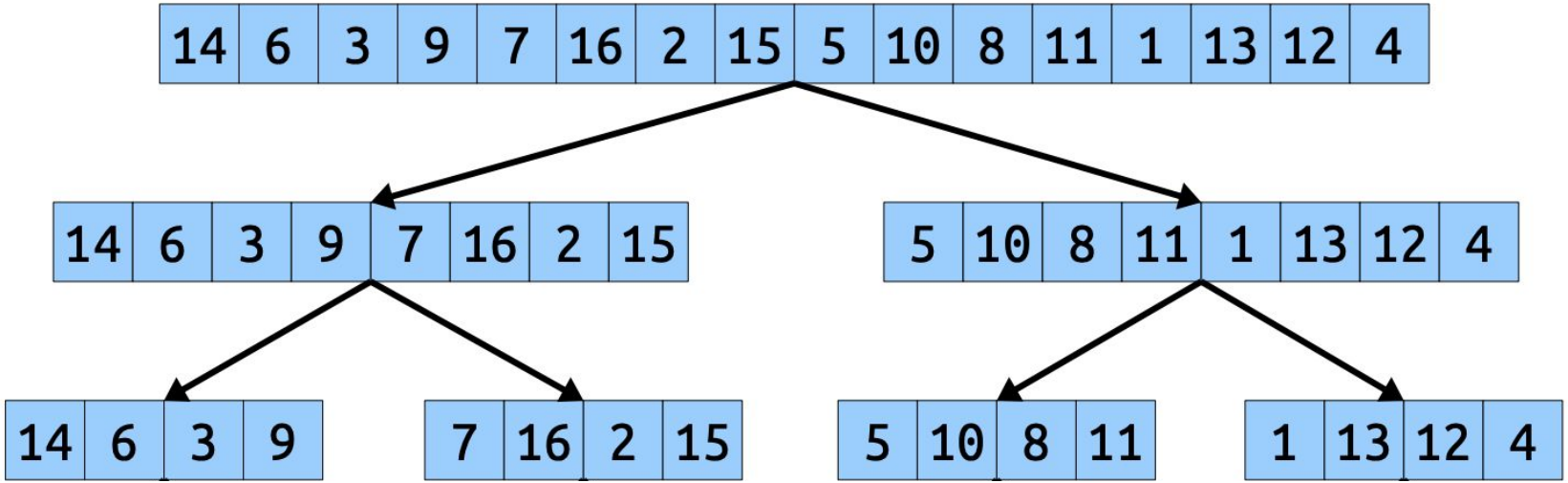
Merge Sort

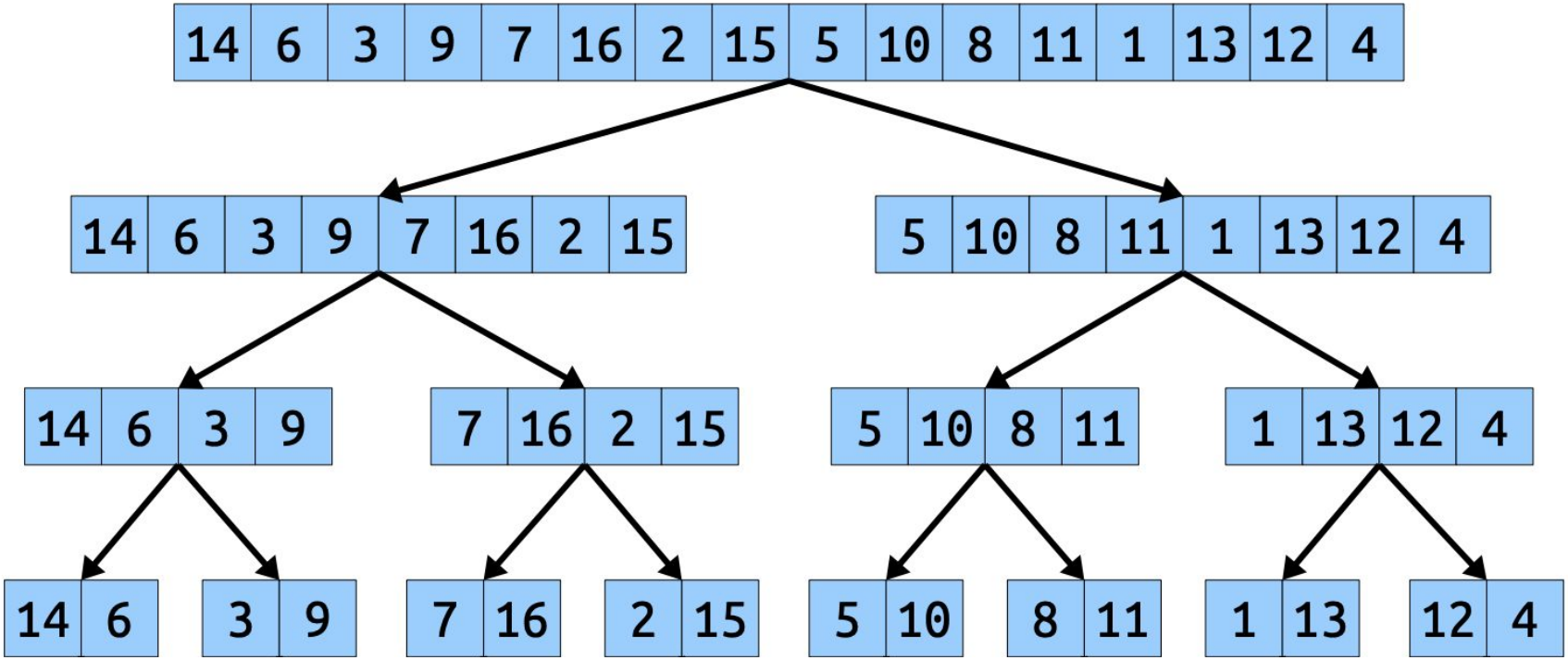
A recursive sorting algorithm!

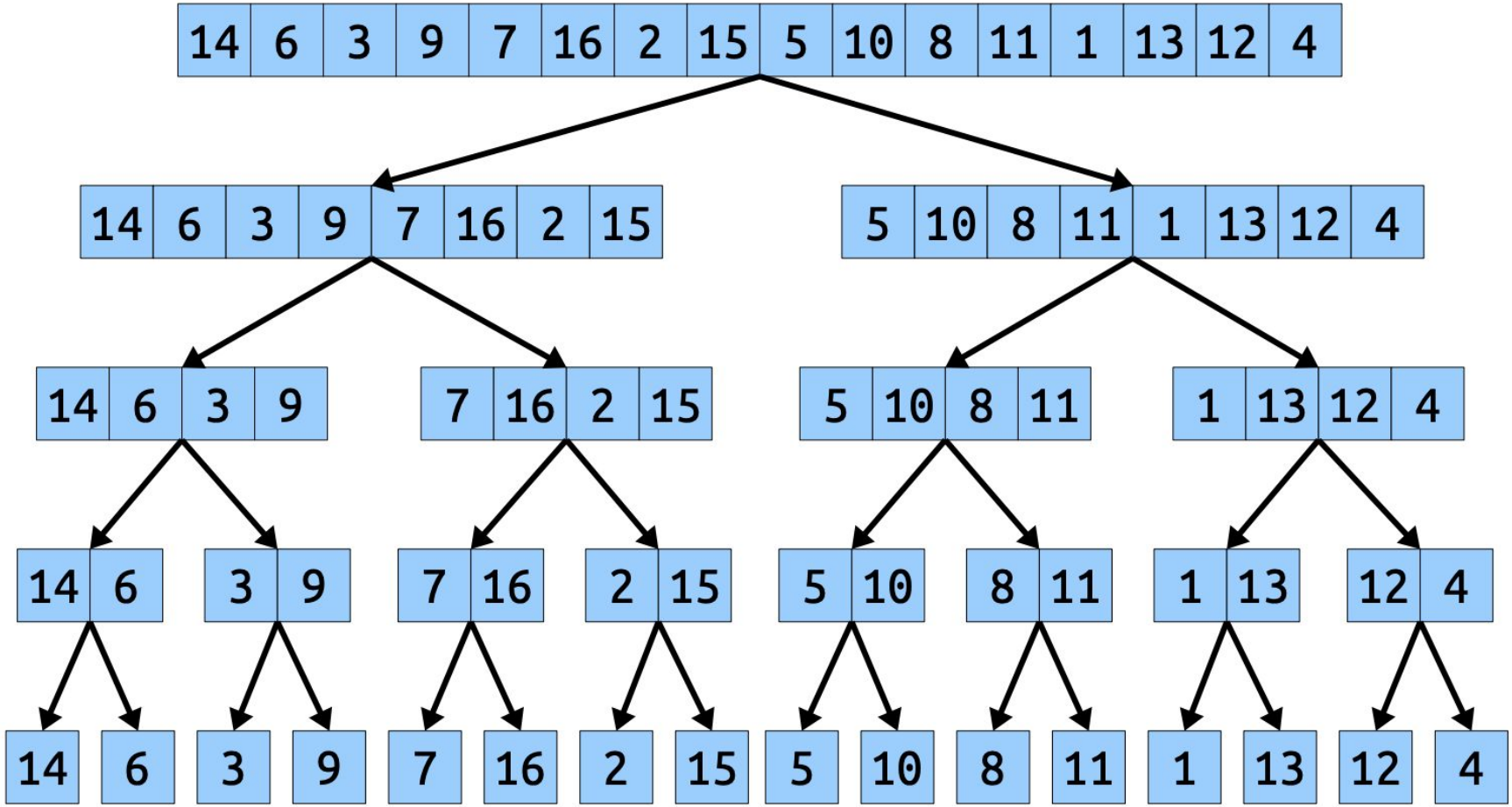
- **Base Case:**
  - An empty or single-element list is already sorted.
- **Recursive step:**
  - Break the list in half and recursively sort each part. (easy divide)
  - Use merge to combine them back into a single sorted list (hard join)

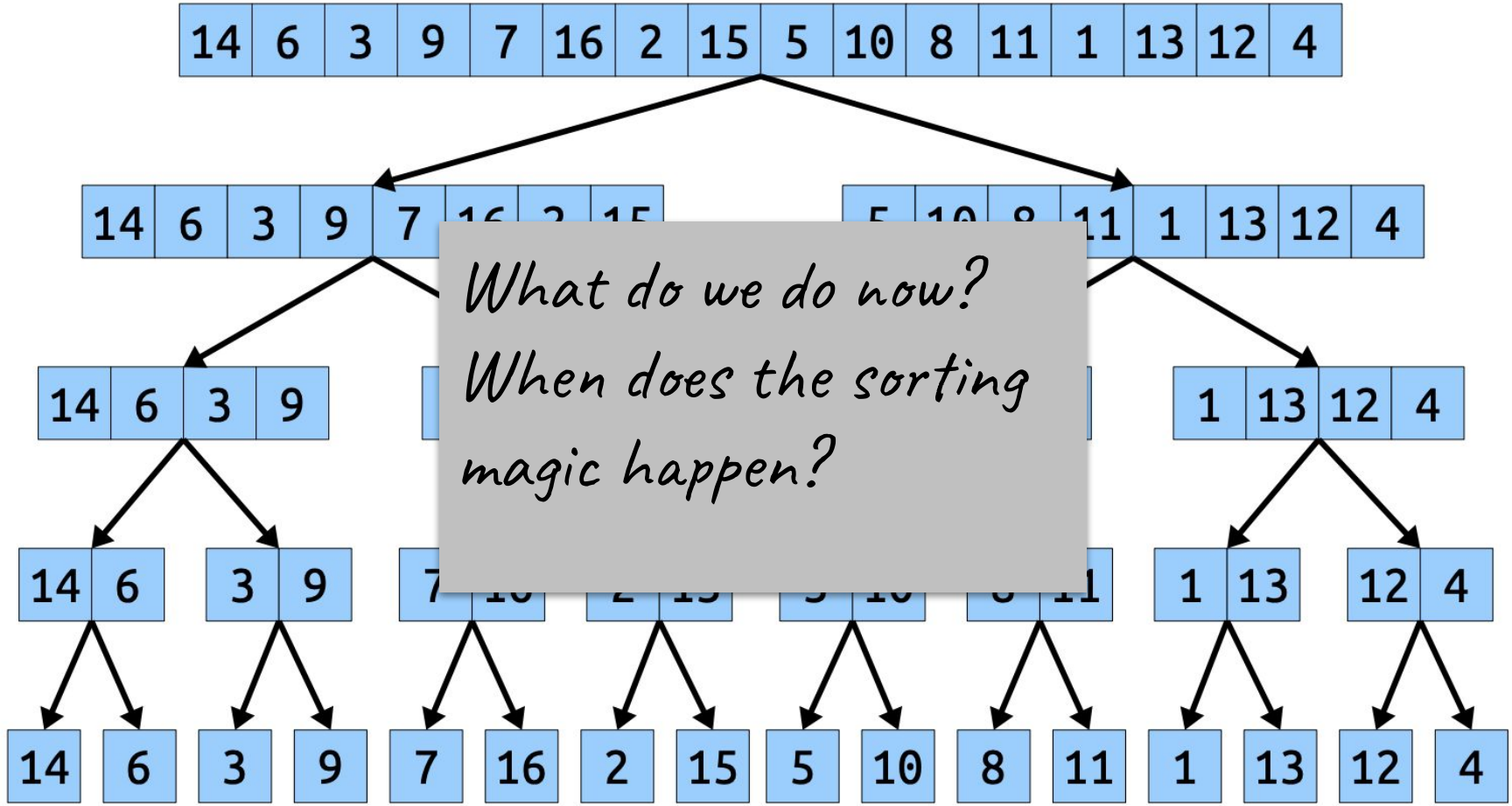

14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4


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

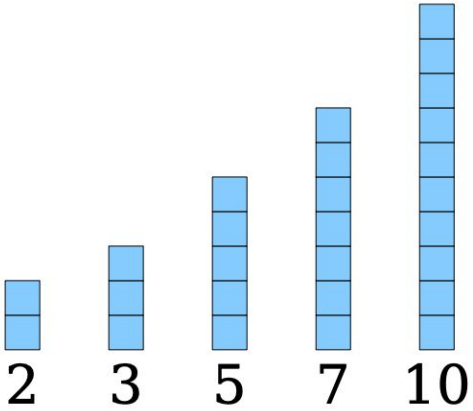

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4


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

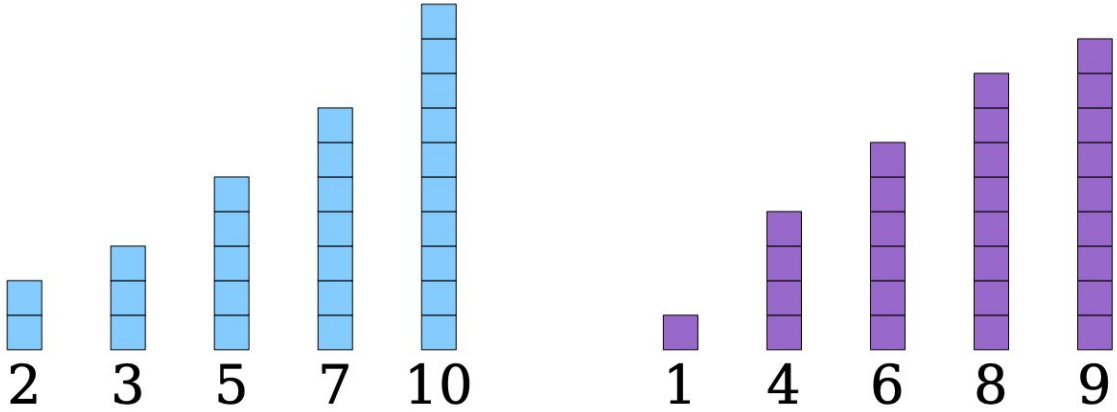

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9
1 13 12 4
14 6
3 9
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4
What do we do now?
When does the sorting magic happen?

The Key Insight: Merge

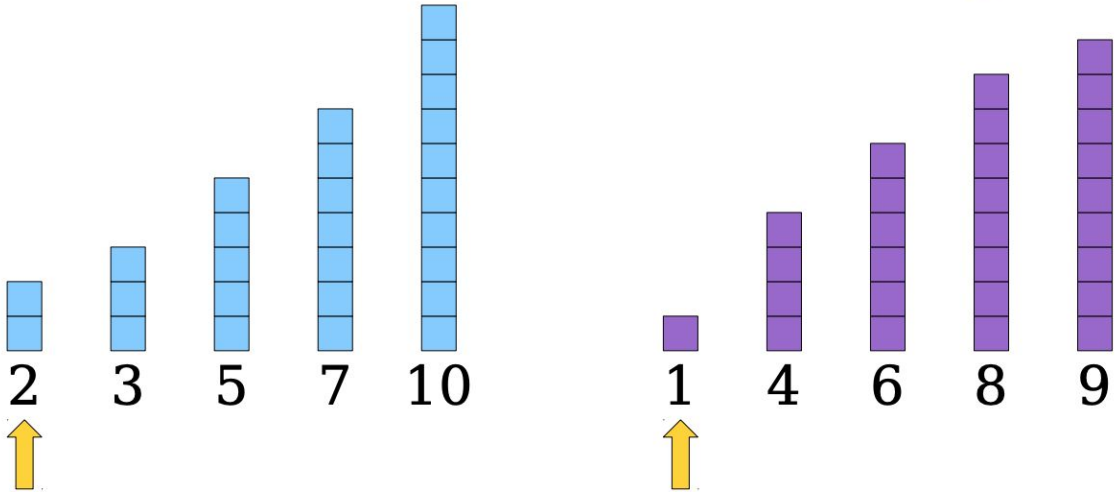
The Key Insight: Merge



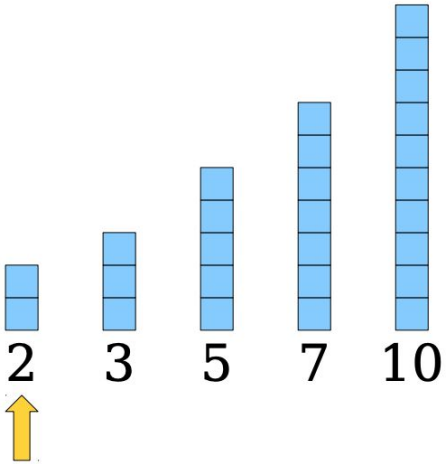
The Key Insight: Merge

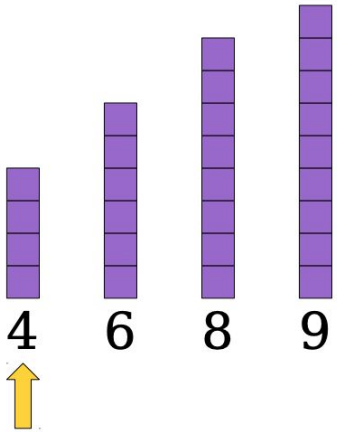


The Key Insight: Merge

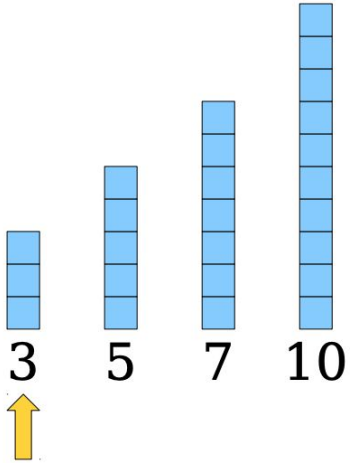


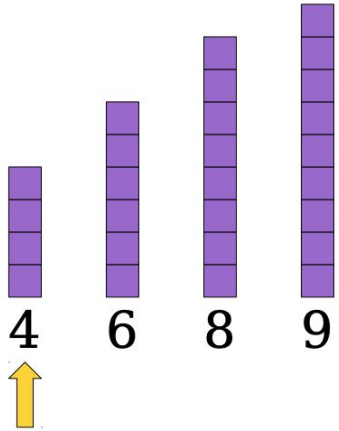
The Key Insight: Merge

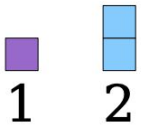




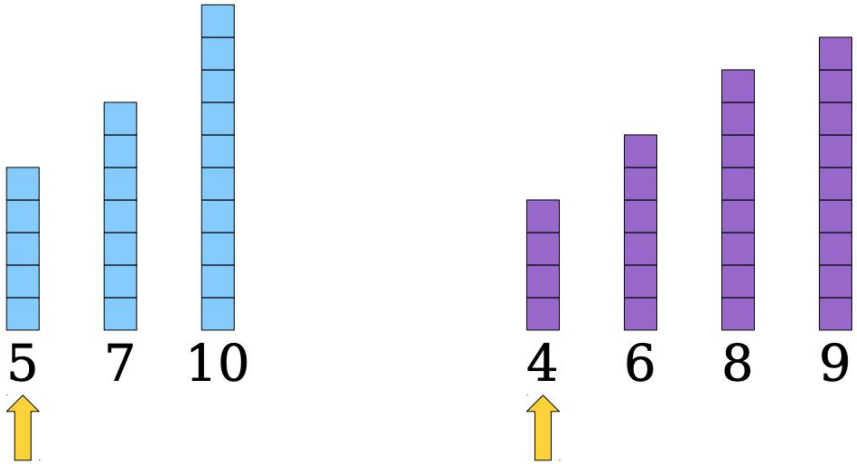
The Key Insight: Merge

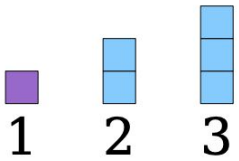




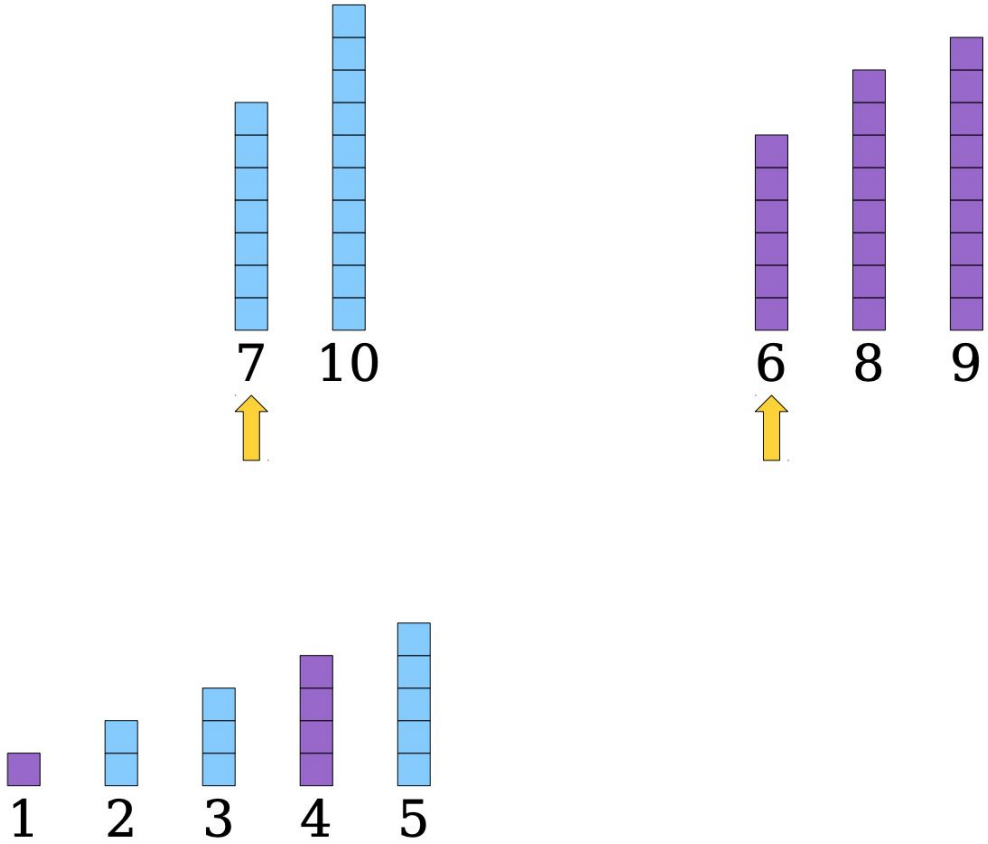


The Key Insight: Merge

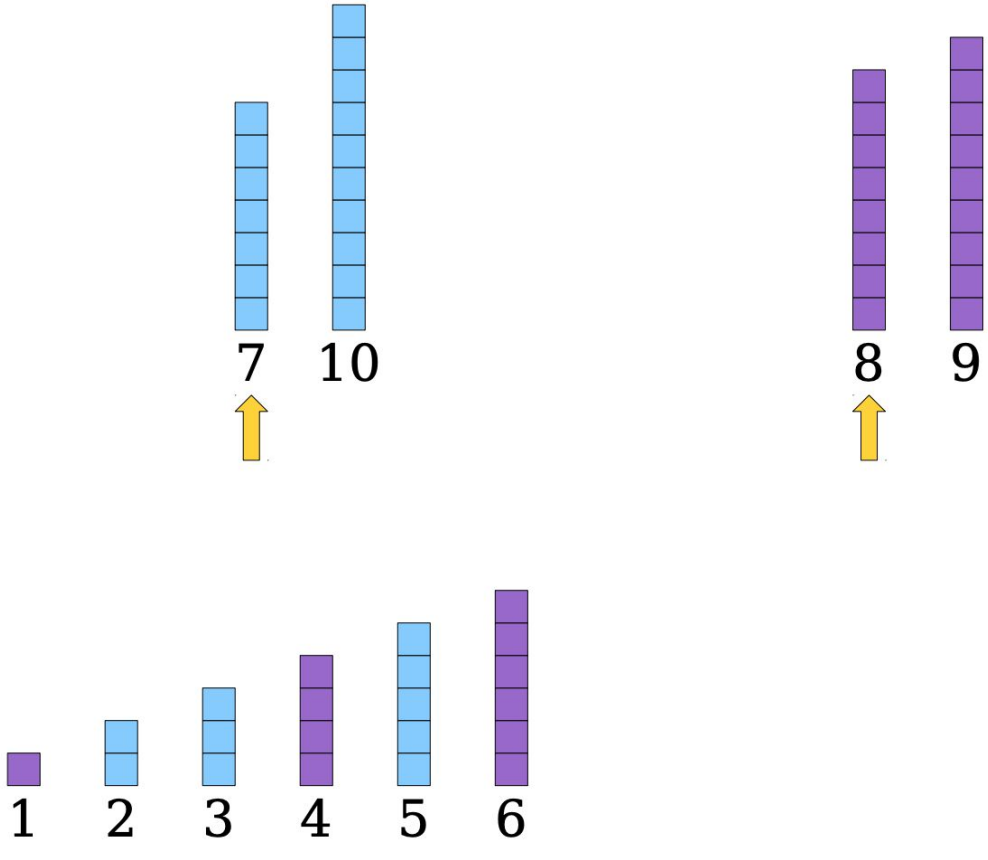




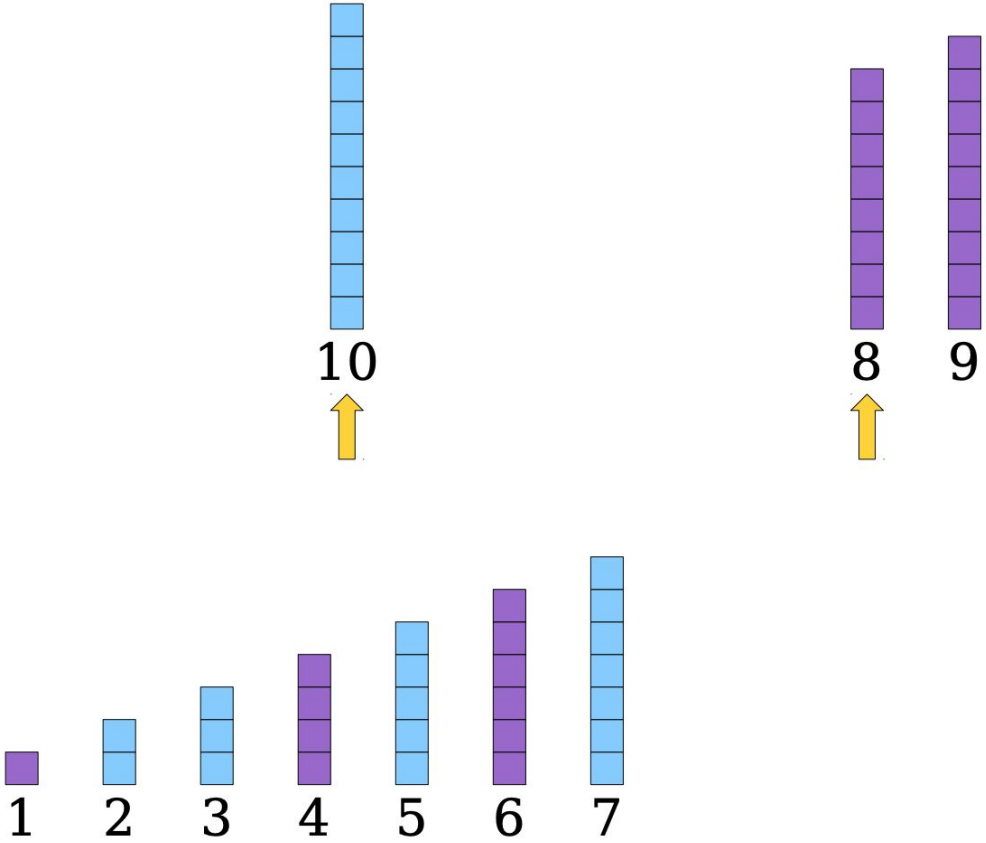
The Key Insight: Merge



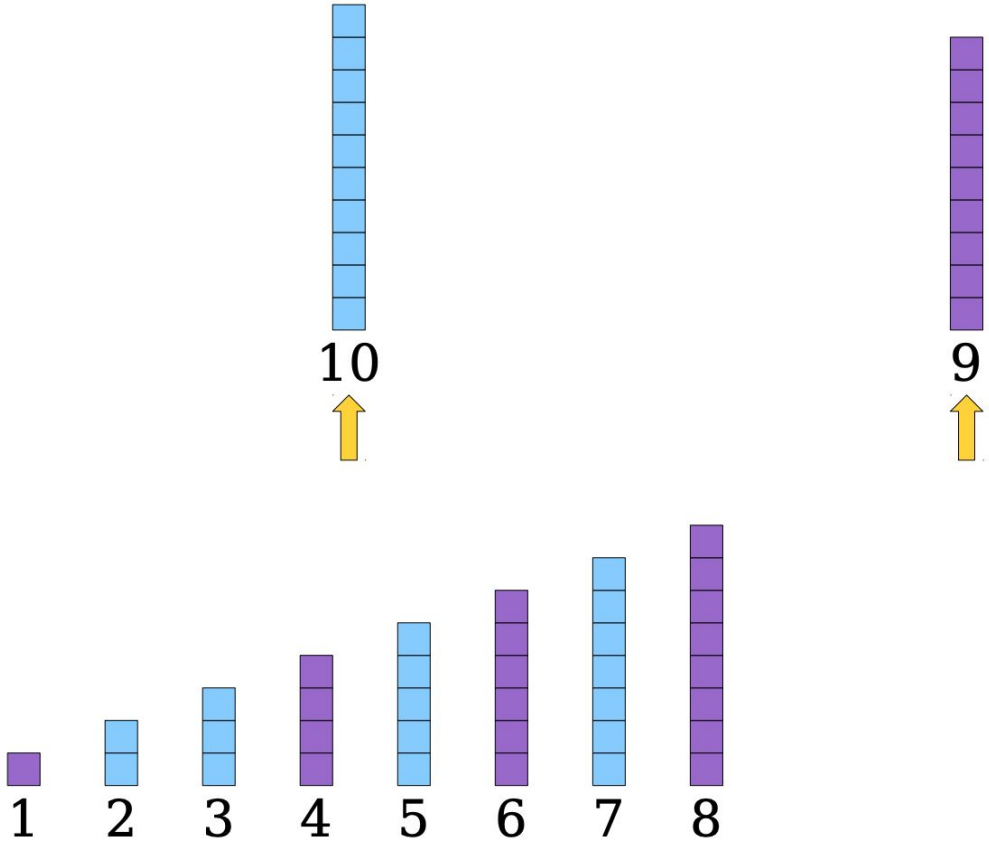
The Key Insight: Merge



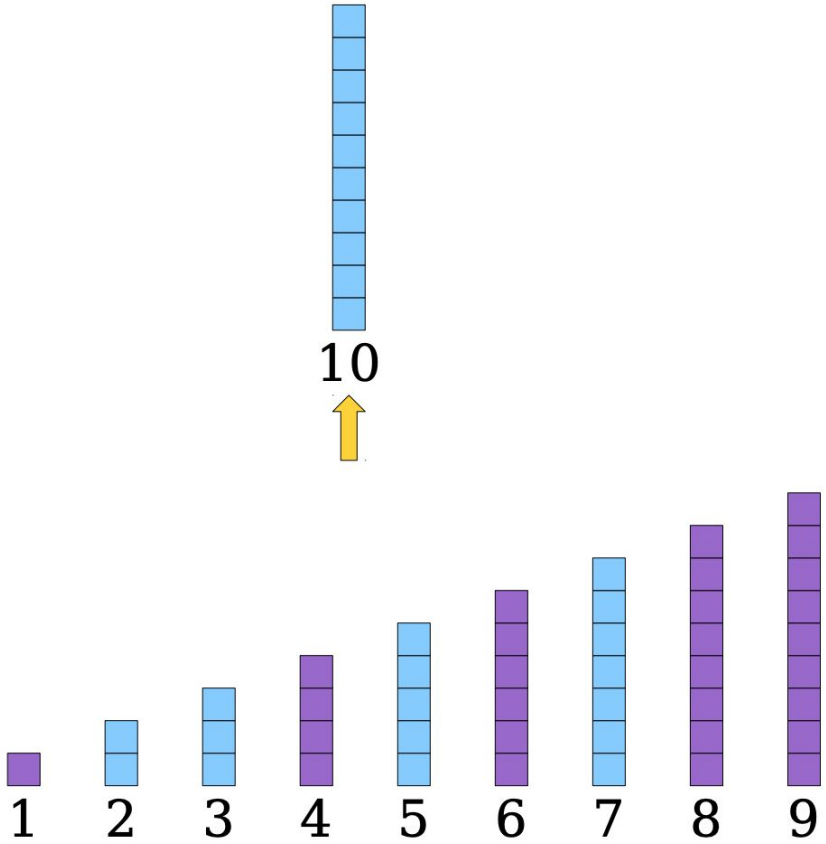
The Key Insight: Merge



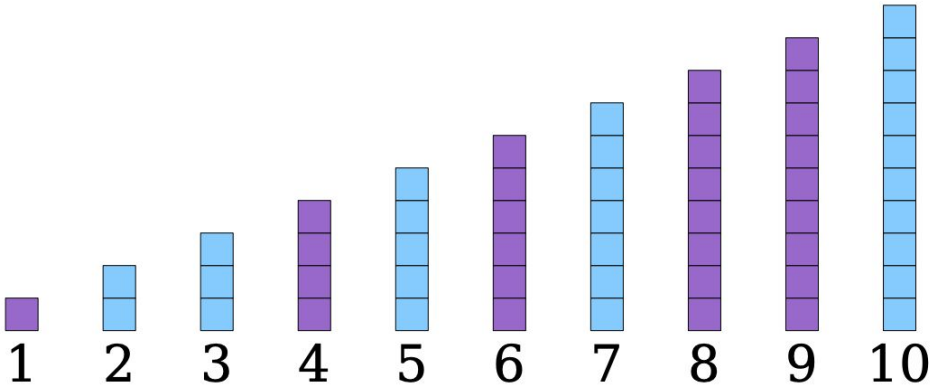
The Key Insight: Merge



The Key Insight: Merge

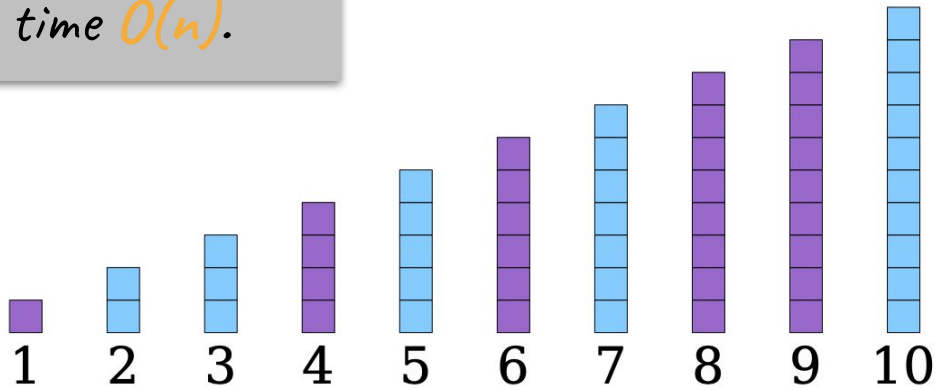


The Key Insight: Merge



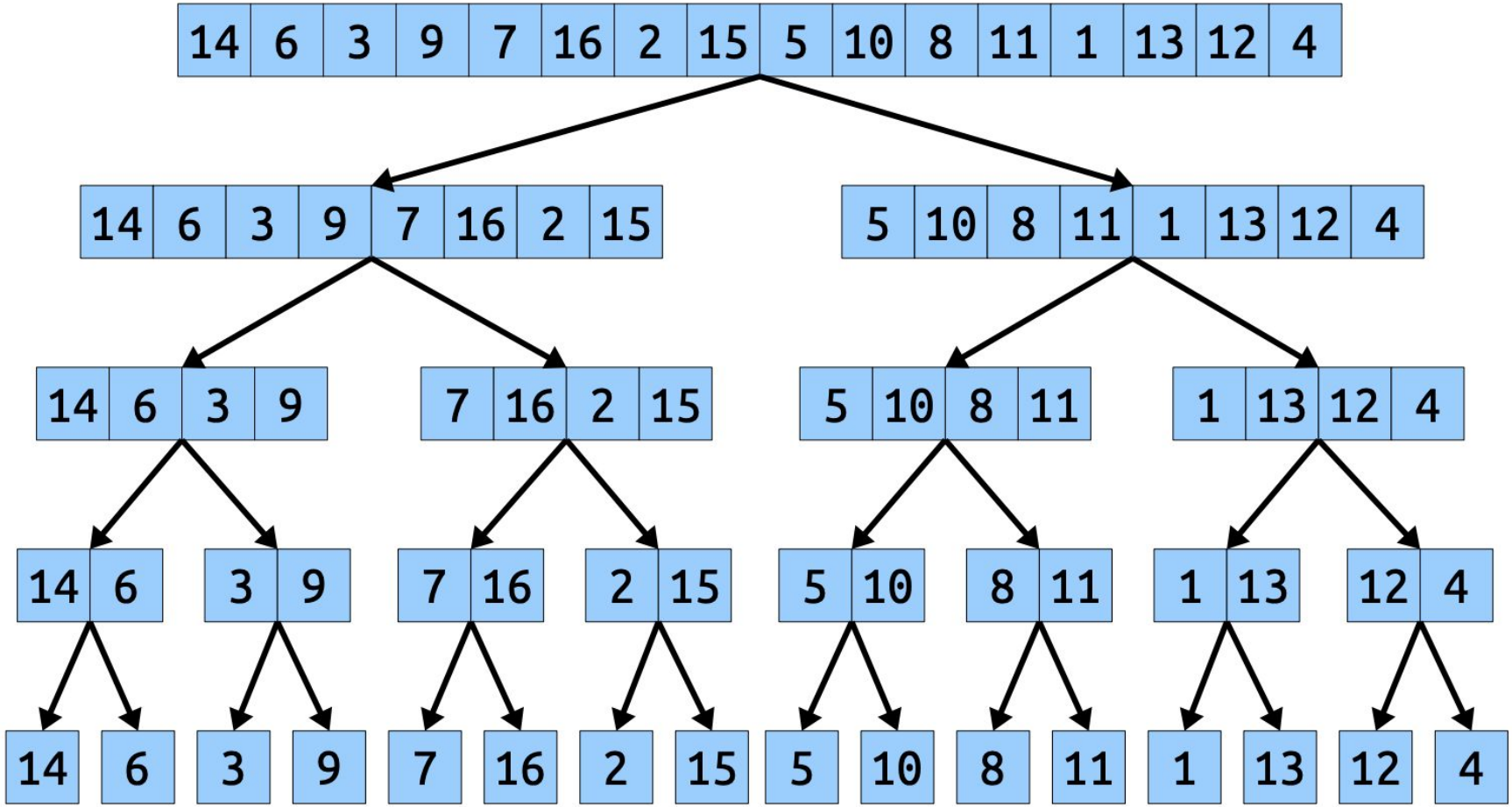
The Key Insight: Merge

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).

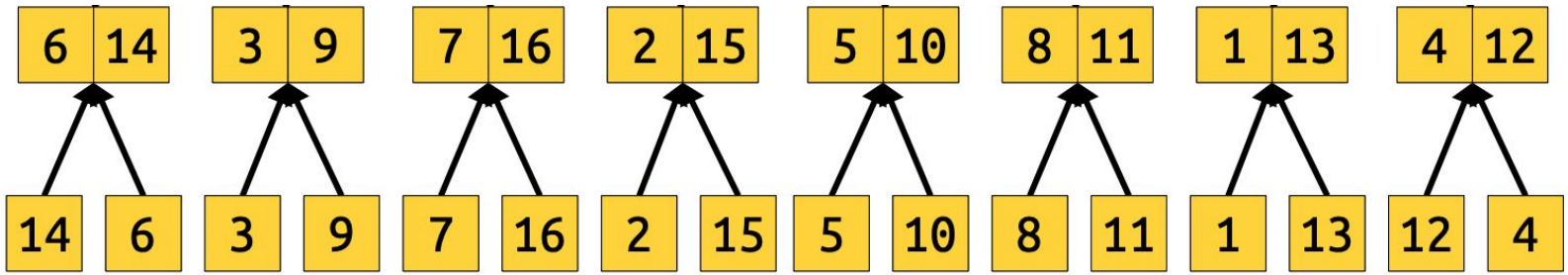


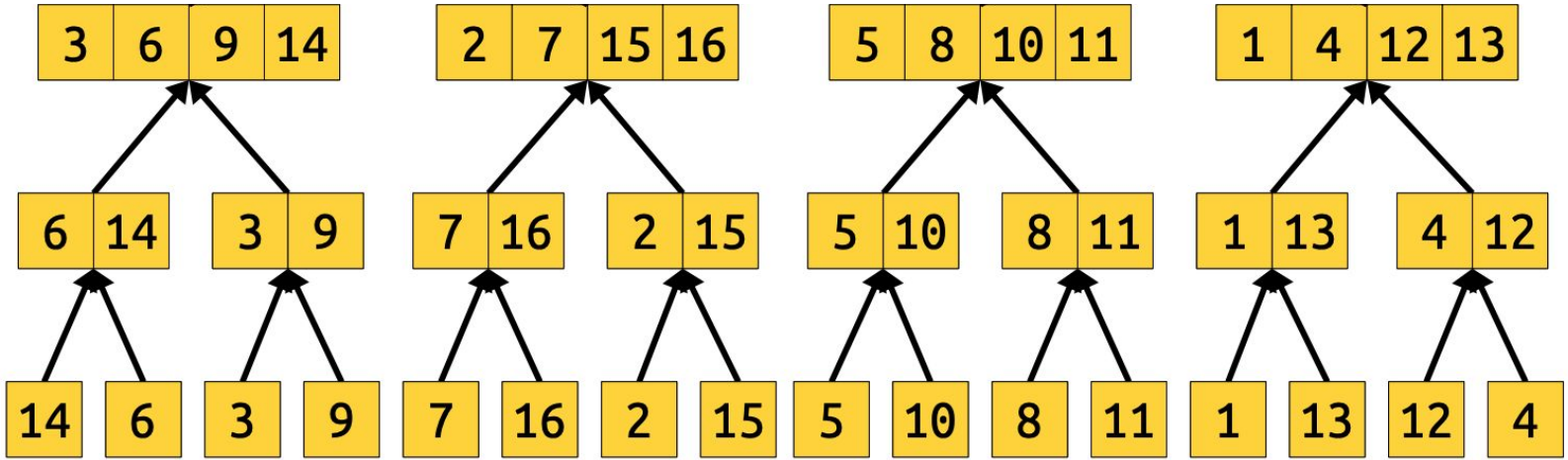
The Key Insight: Merge

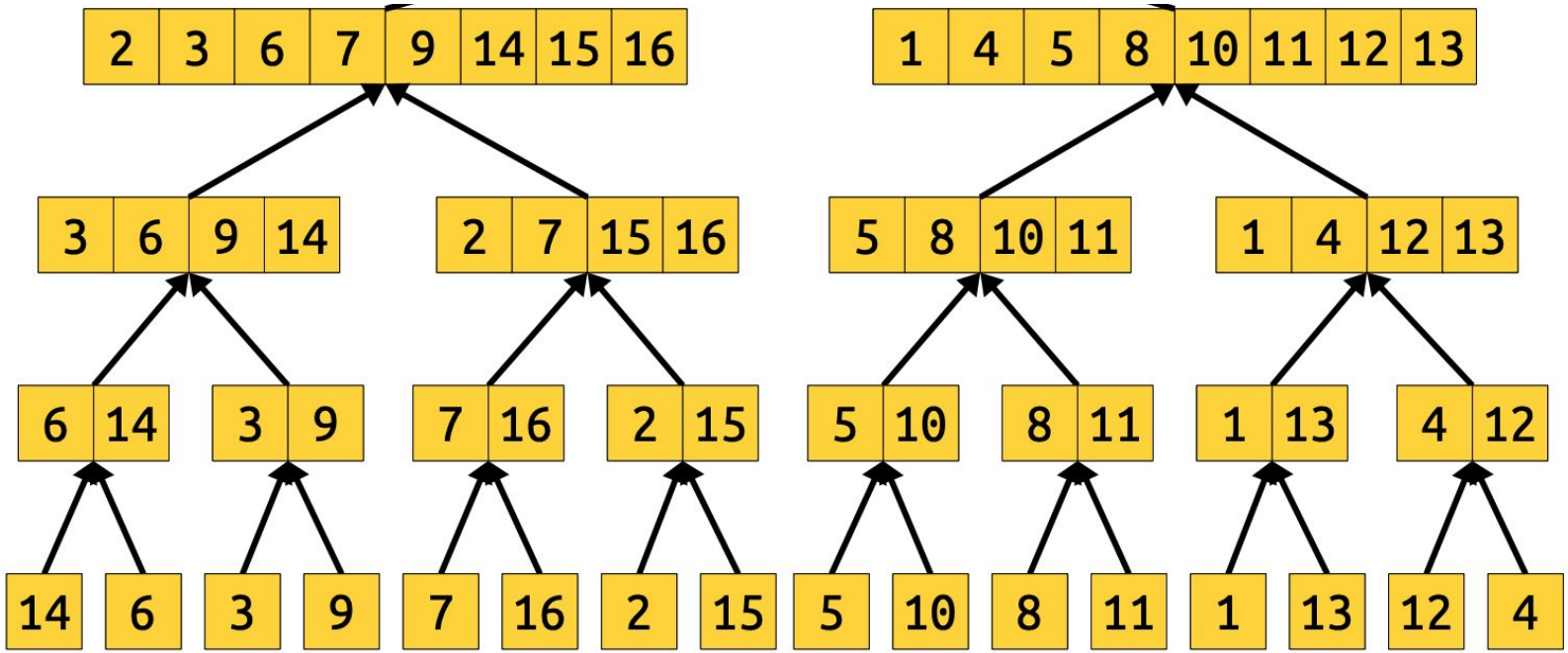
- The merge algorithm takes in two sorted lists and combines them into a single sorted list.
- While both lists are nonempty, compare their first elements. Remove the smaller element and append it to the output.
- Once one list is empty, add all elements from the other list to the output.

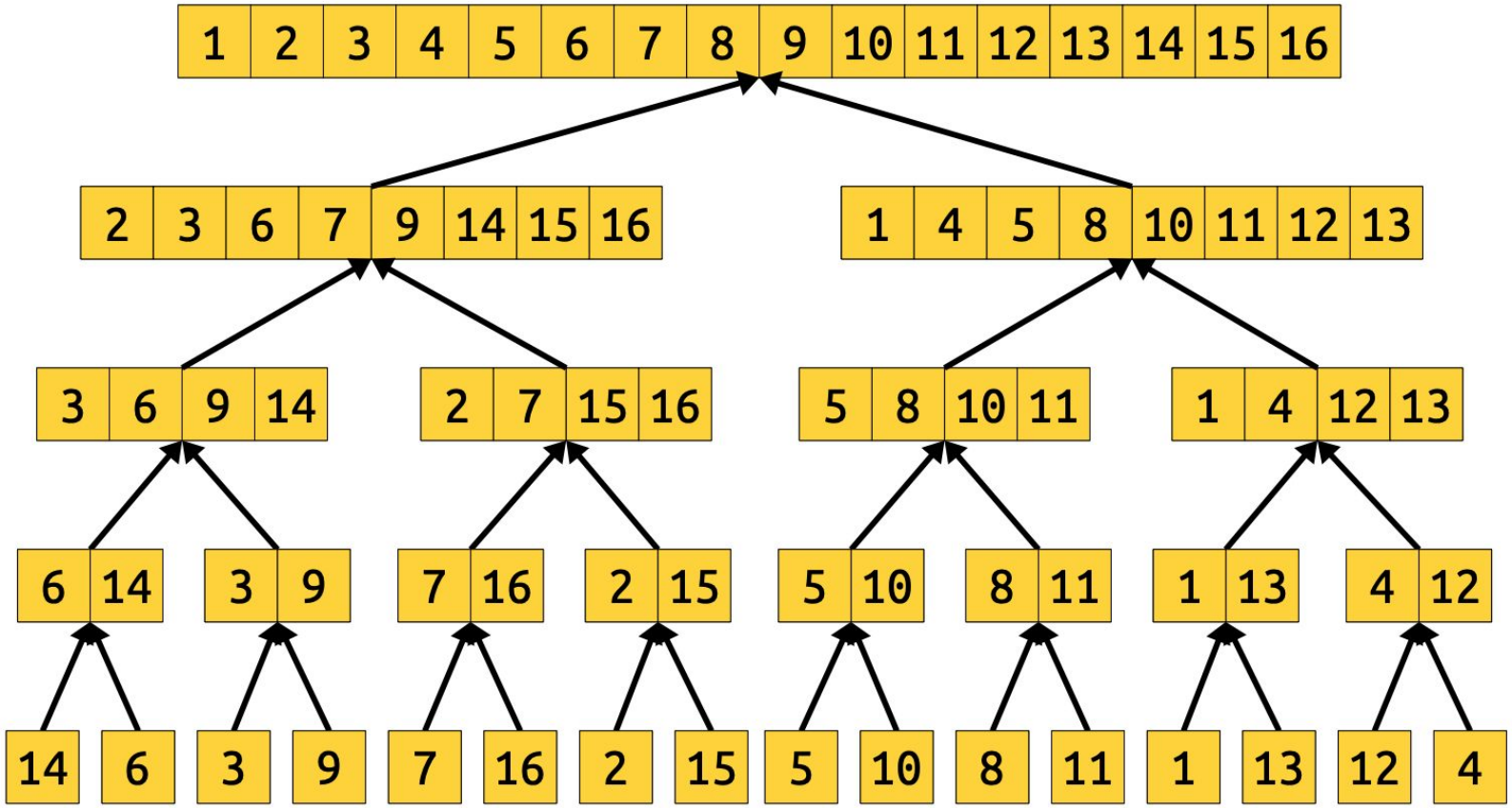

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

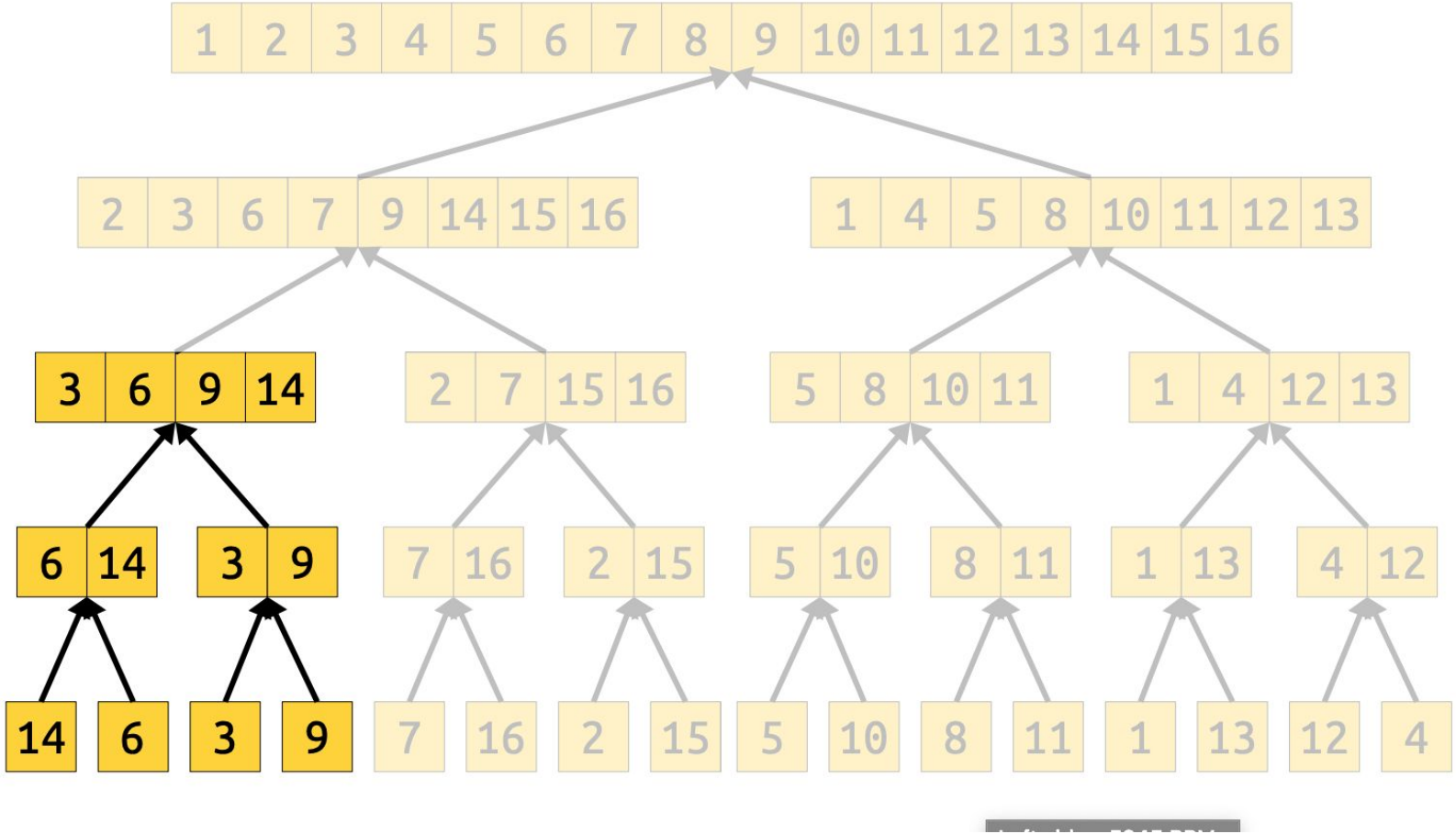

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4


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!

Announcements

Announcements

- Revisions for Assignment 4 will be due **today at 11:59pm PDT.**

- Assignment 6 has been released and is due on **Friday, August 13 at 11:59pm PDT with a 24-hour grace period**.

- Final project feedback will be released by Wednesday morning.
  - If you received a comment asking you to meet with Nick or me, please come see us during OH!
  - If you decide to change your project topic from your proposal, we also recommend checking in with one of us about your new idea.
  - Feel free to come chat with us if you have any questions.

Analyzing Mergesort:

How fast is this sorting algorithm?

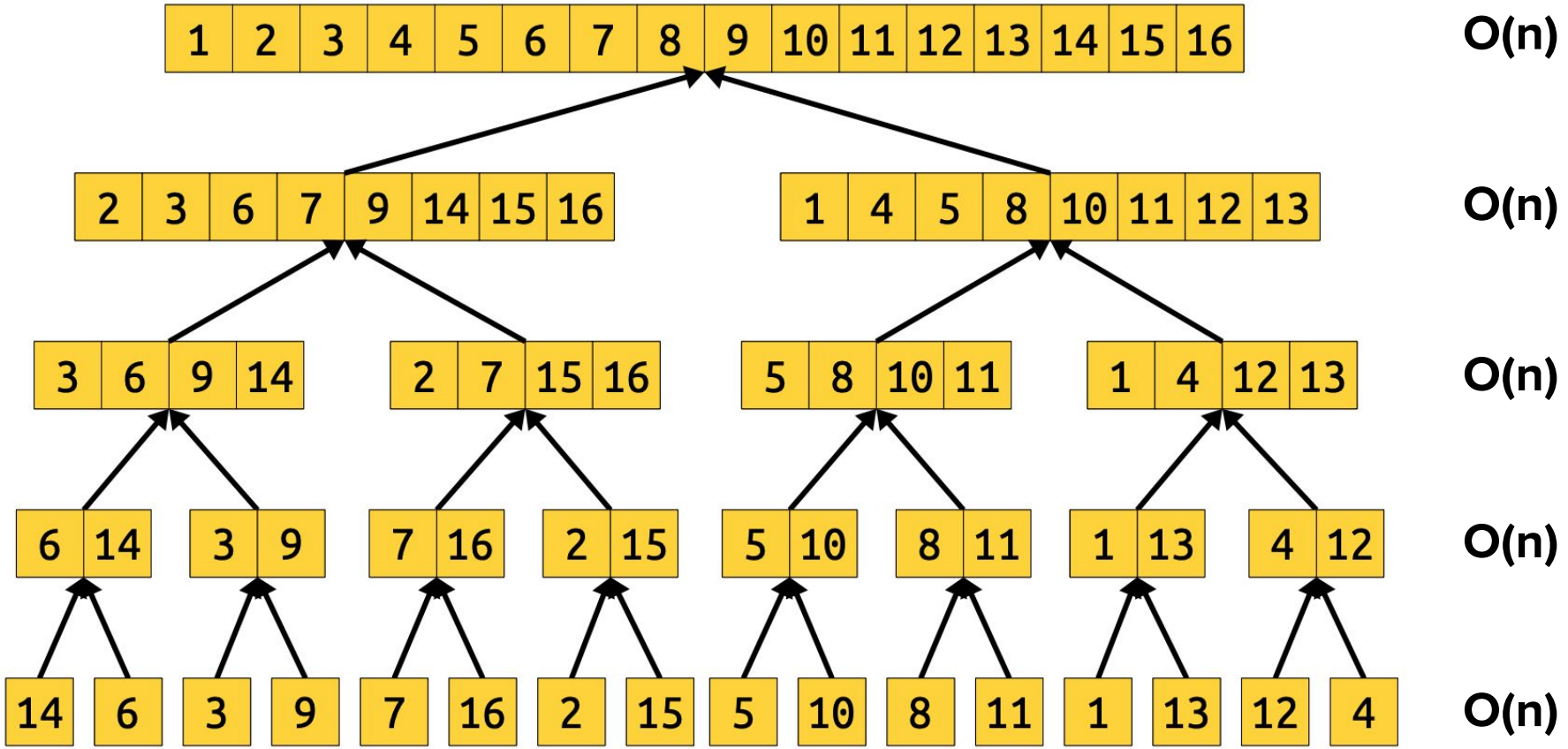
```
void mergeSort(Vector<int>& vec) {
   /* A list with 0 or 1 elements is already sorted by definition. */
   if (vec.size() <= 1) return;

   /* 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);
}
```

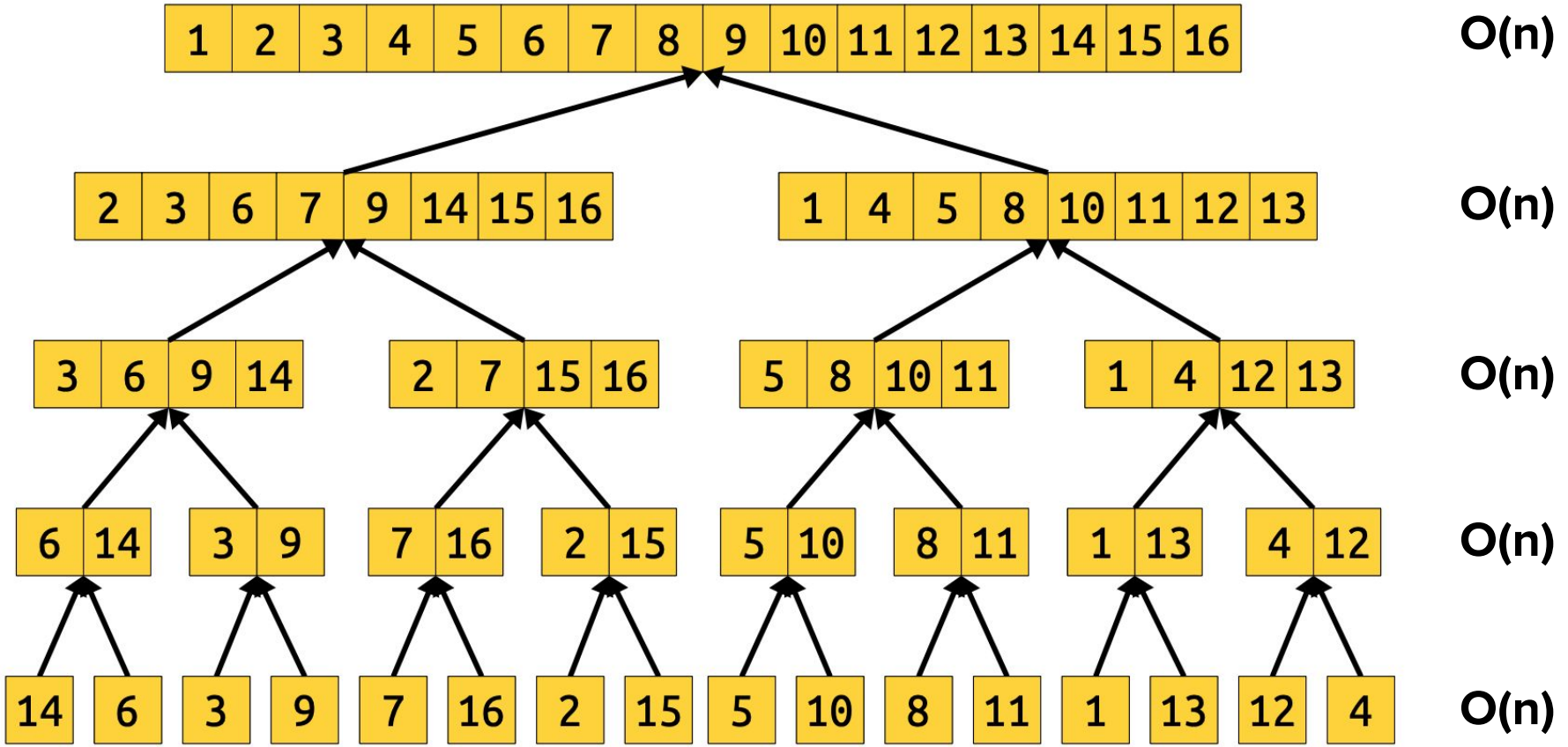
O(n) work

O(n) work

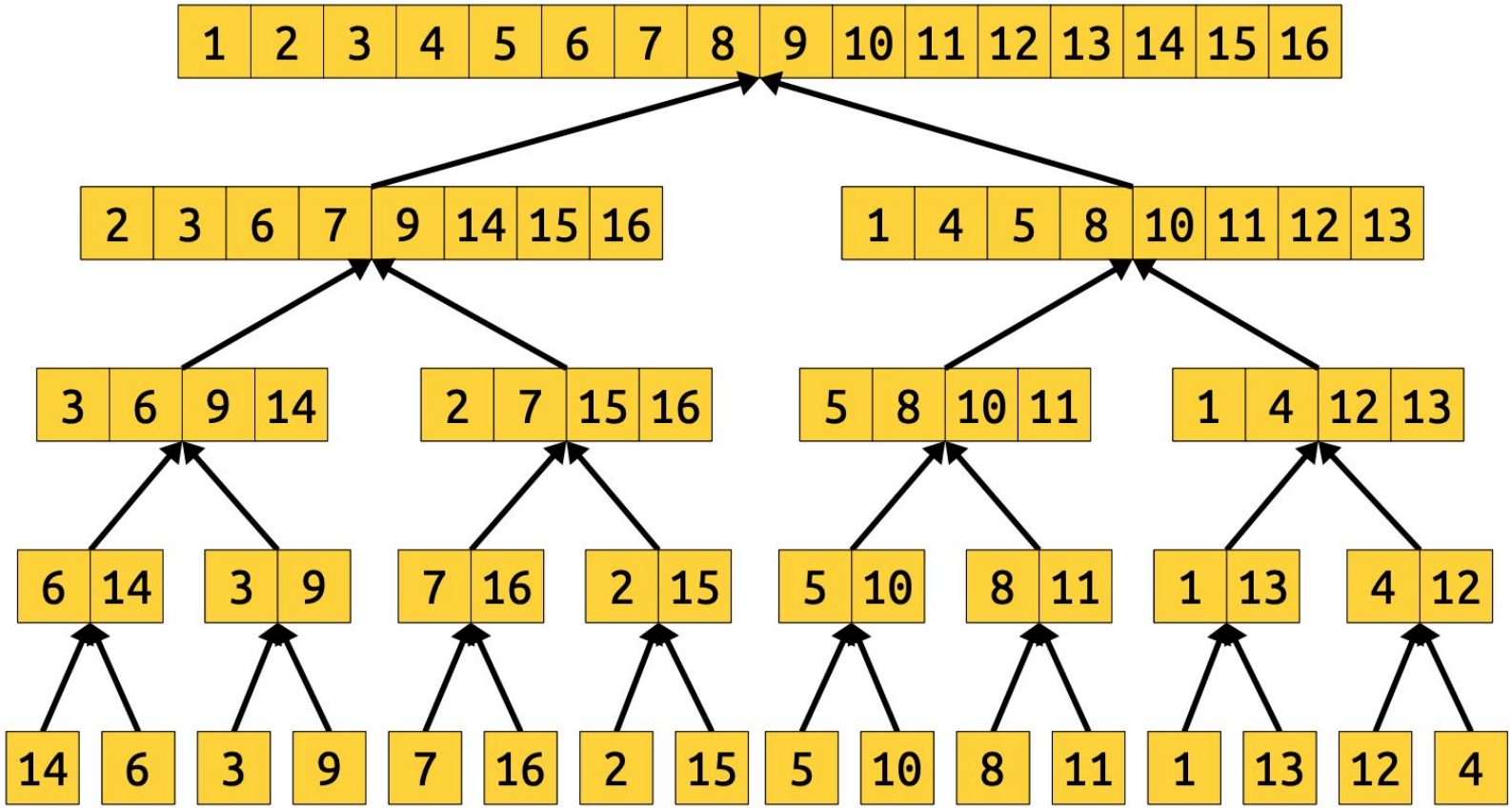
```
void mergeSort(Vector<int>& vec) {
   /* A list with 0 or 1 elements is already sorted by definition. */
   if (vec.size() <= 1) return;

   /* 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);
}
```



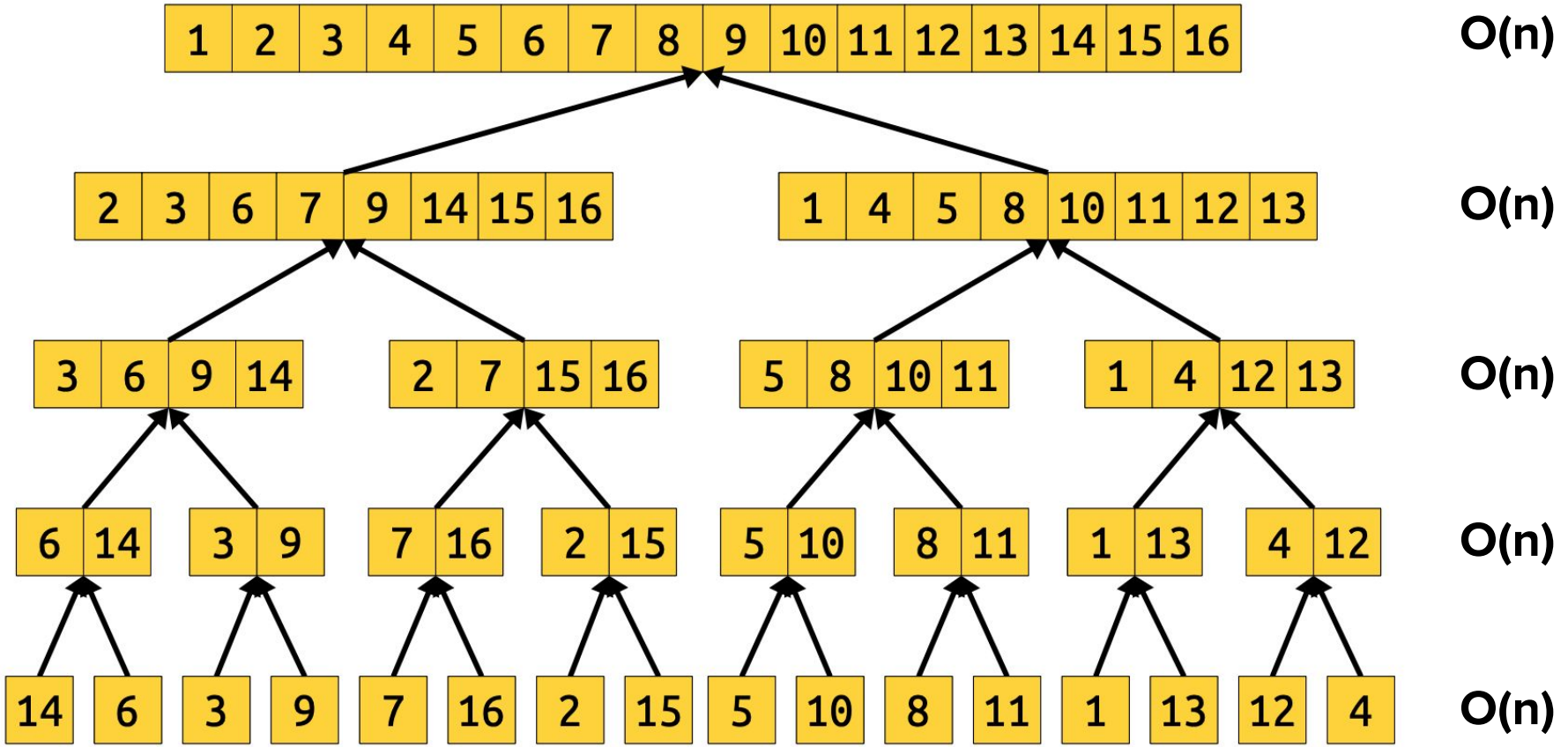
O(n) work at each level!



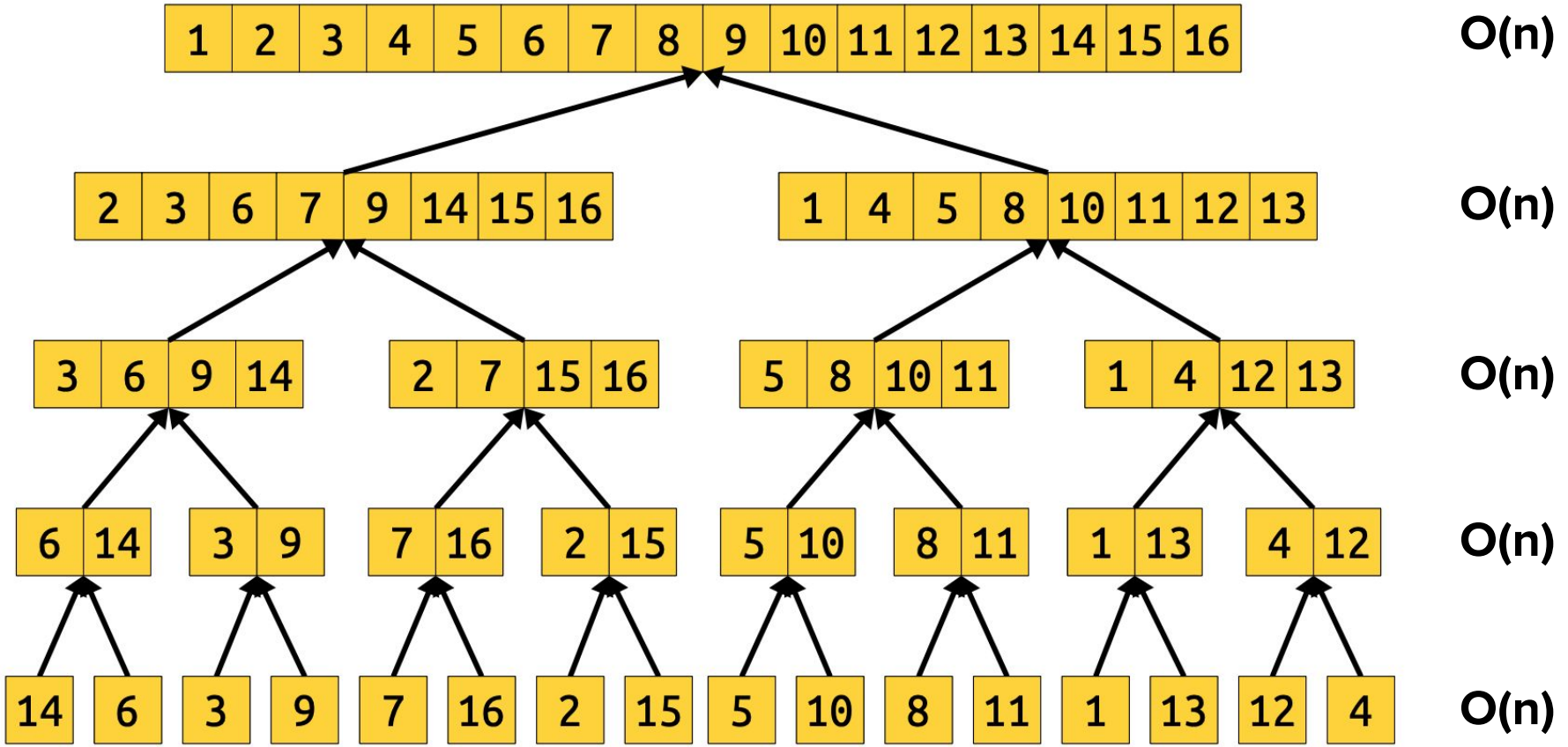
How many levels are there?



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?
  - Let's explore one more divide-and-conquer sort!

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.

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.

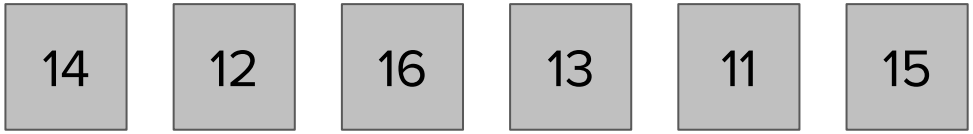
Quicksort Algorithm

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

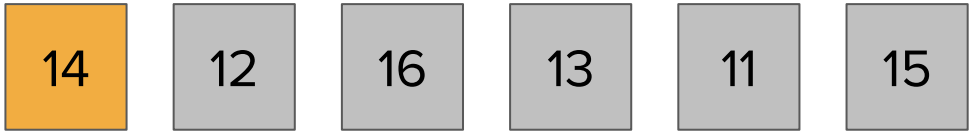
2. **Recursively sort** the two partitions that are not equal to the pivot (smaller and larger elements).
   - Now our smaller elements are in sorted order, and our larger elements are also in sorted order!

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

Input of unsorted elements:

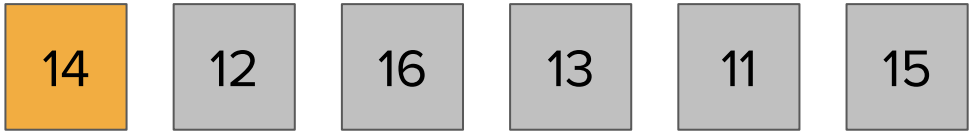


Input of unsorted elements:



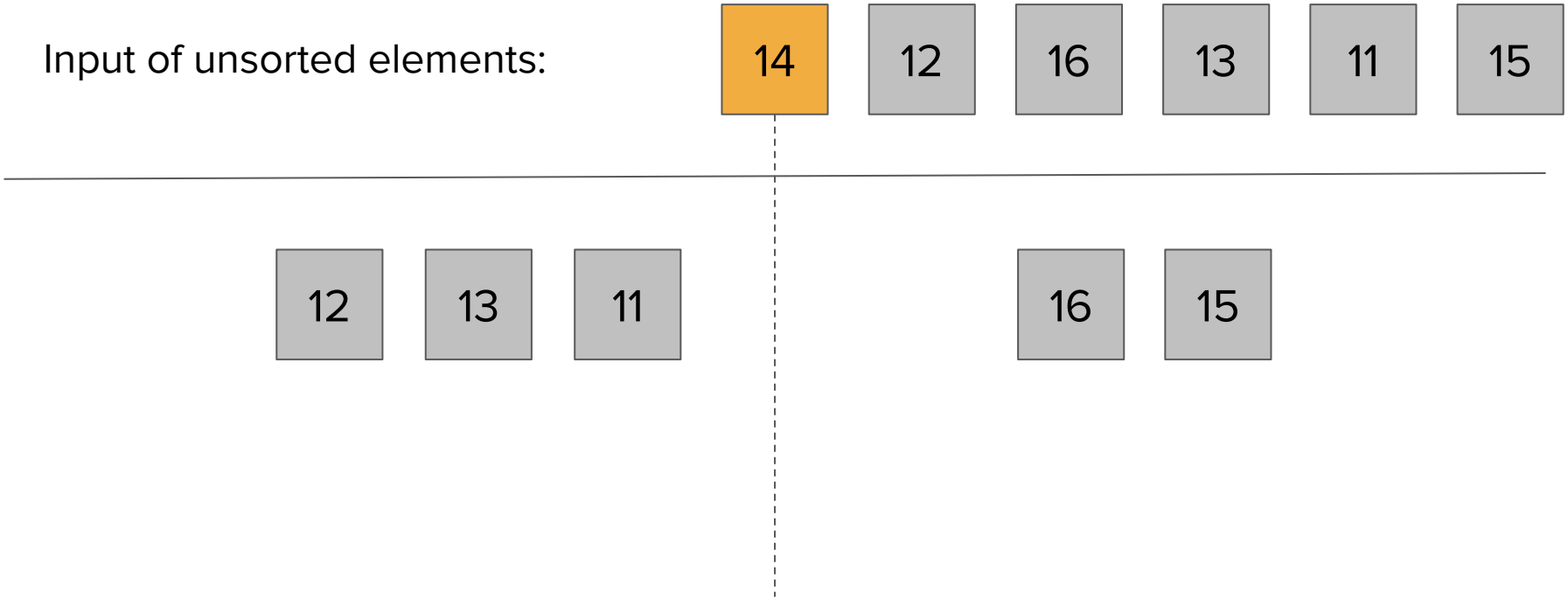
Choose the first element as the pivot.

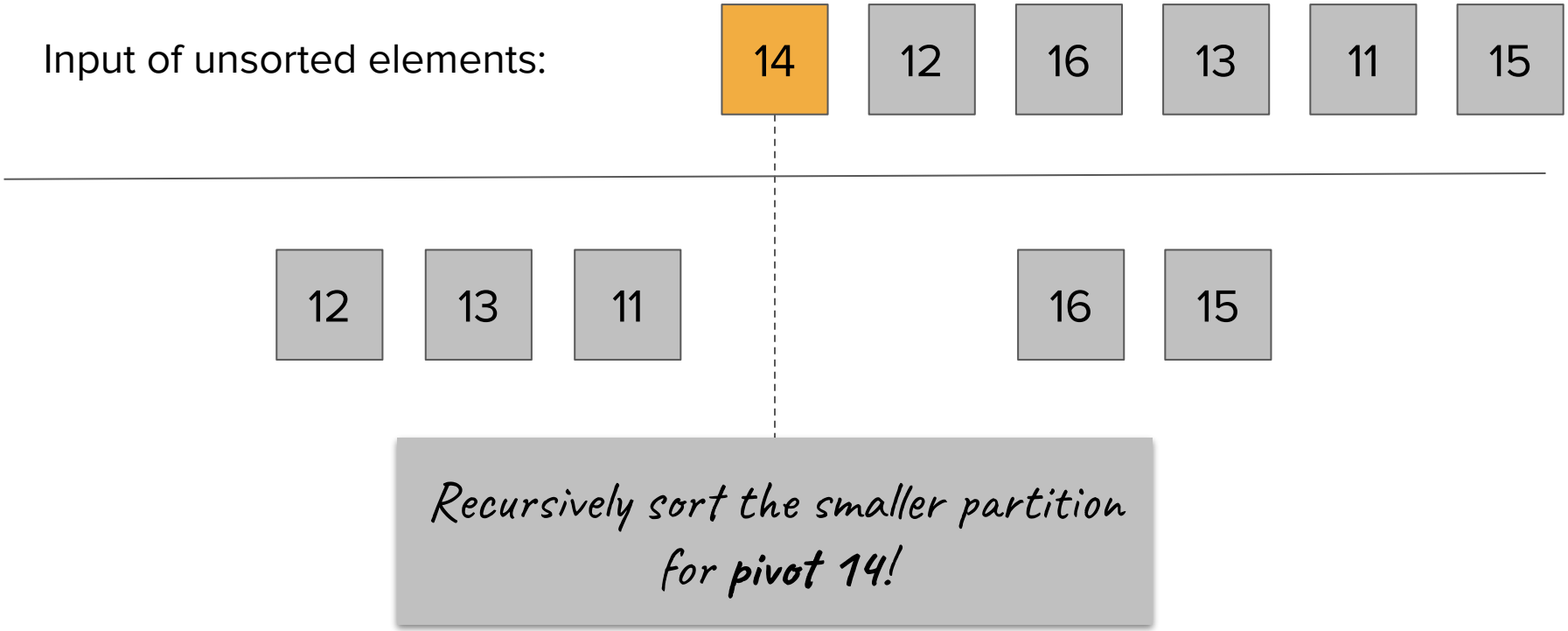
Input of unsorted elements:



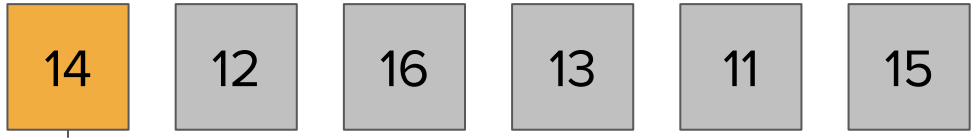
Partition elements into smaller than, equal to, and greater than the pivot.

Input of unsorted elements:



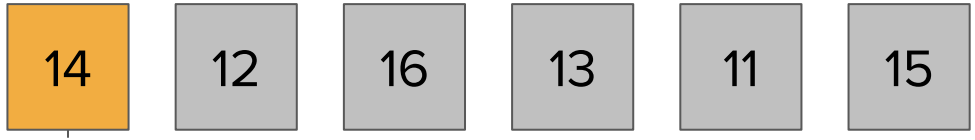

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
Recursively sort the smaller partition
for pivot 14!

Input of unsorted elements:

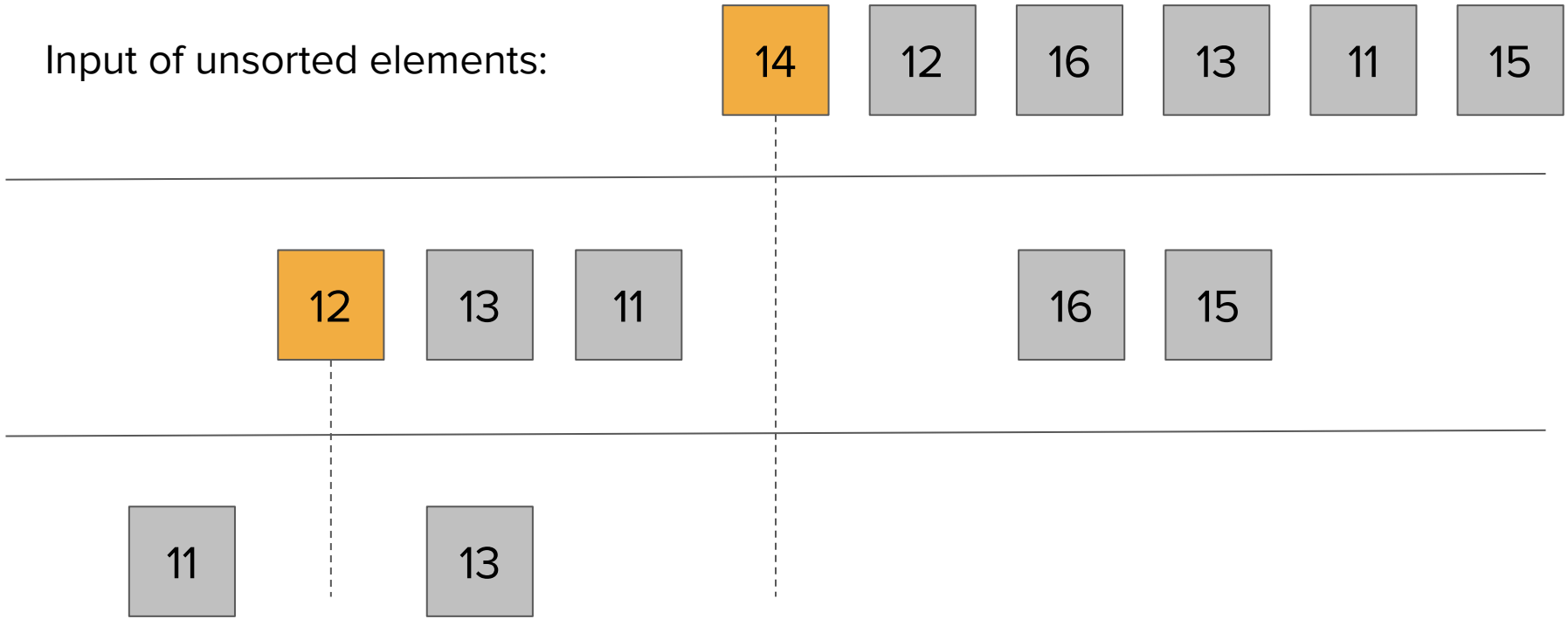


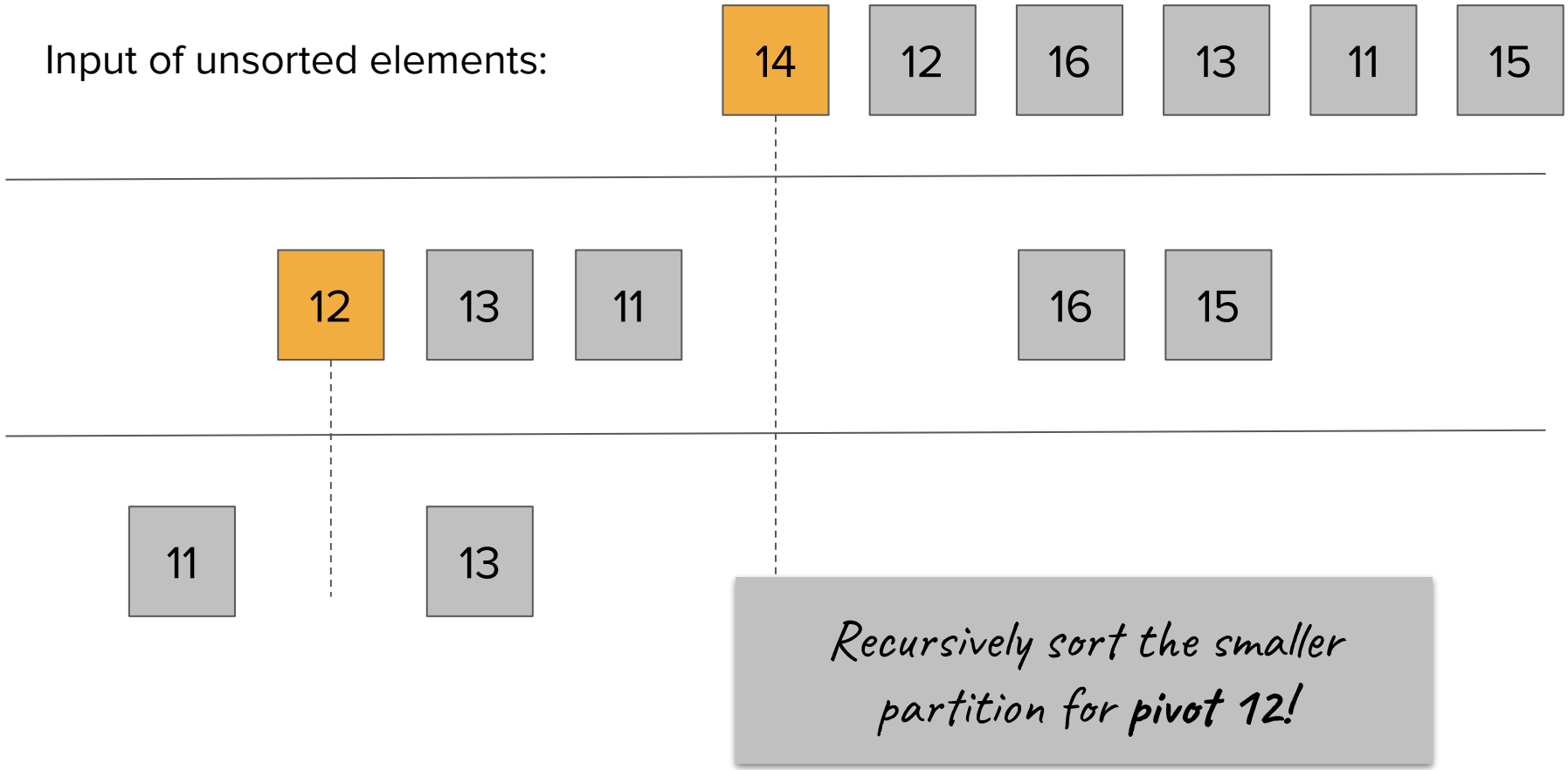
Choose the first element as the pivot.

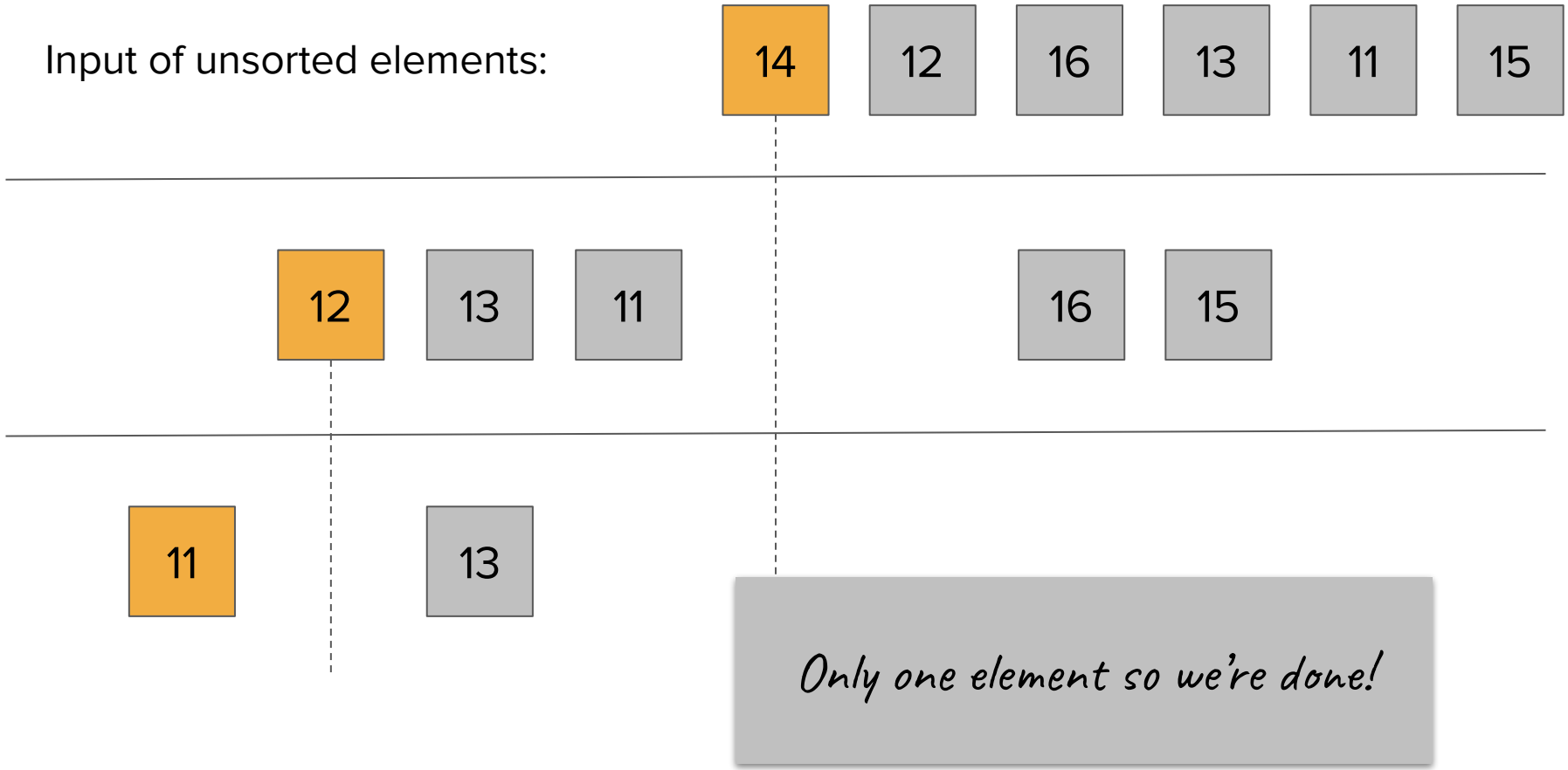
Input of unsorted elements:

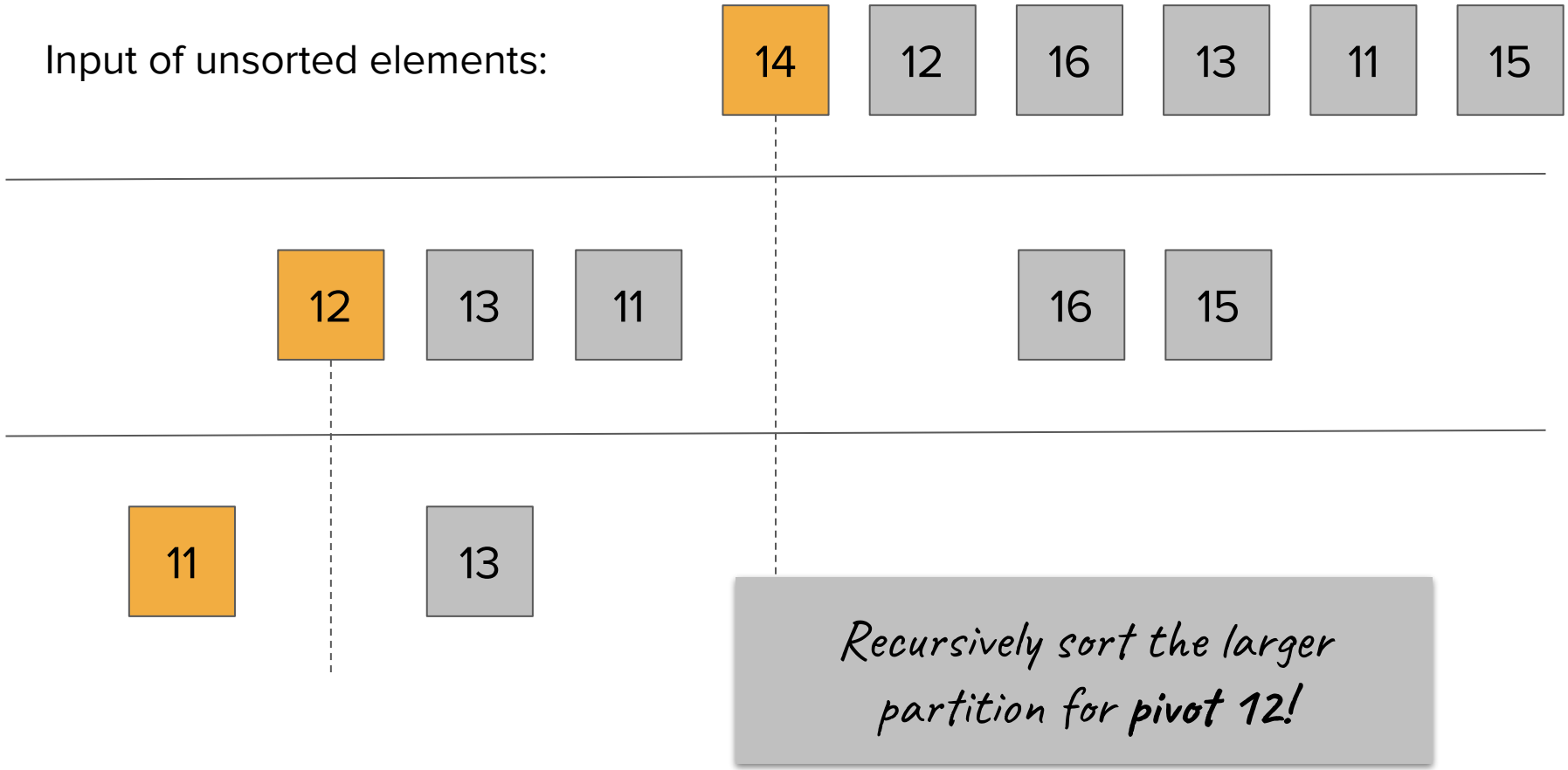


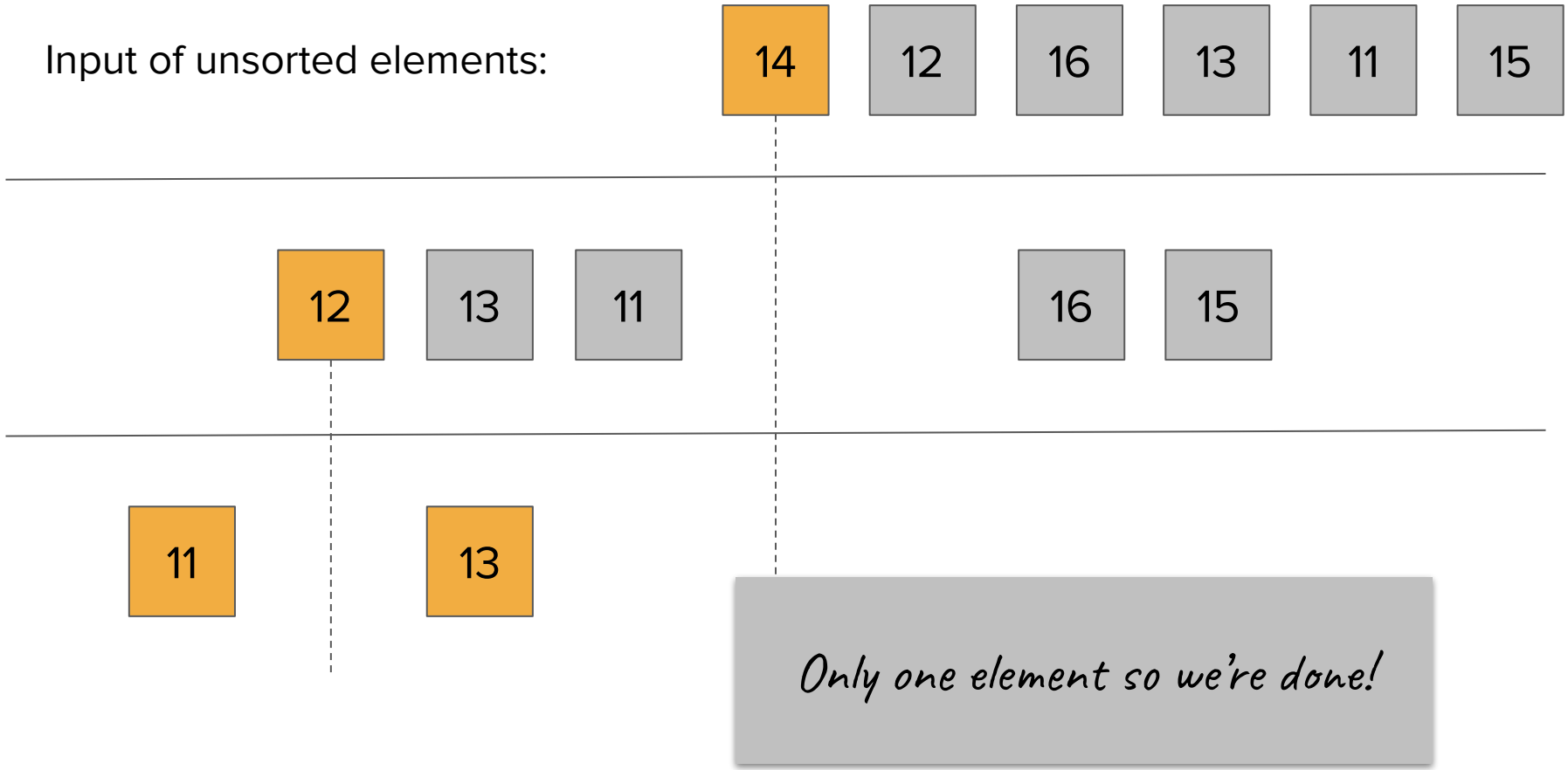
Partition elements into smaller than, equal to, and greater than the pivot.

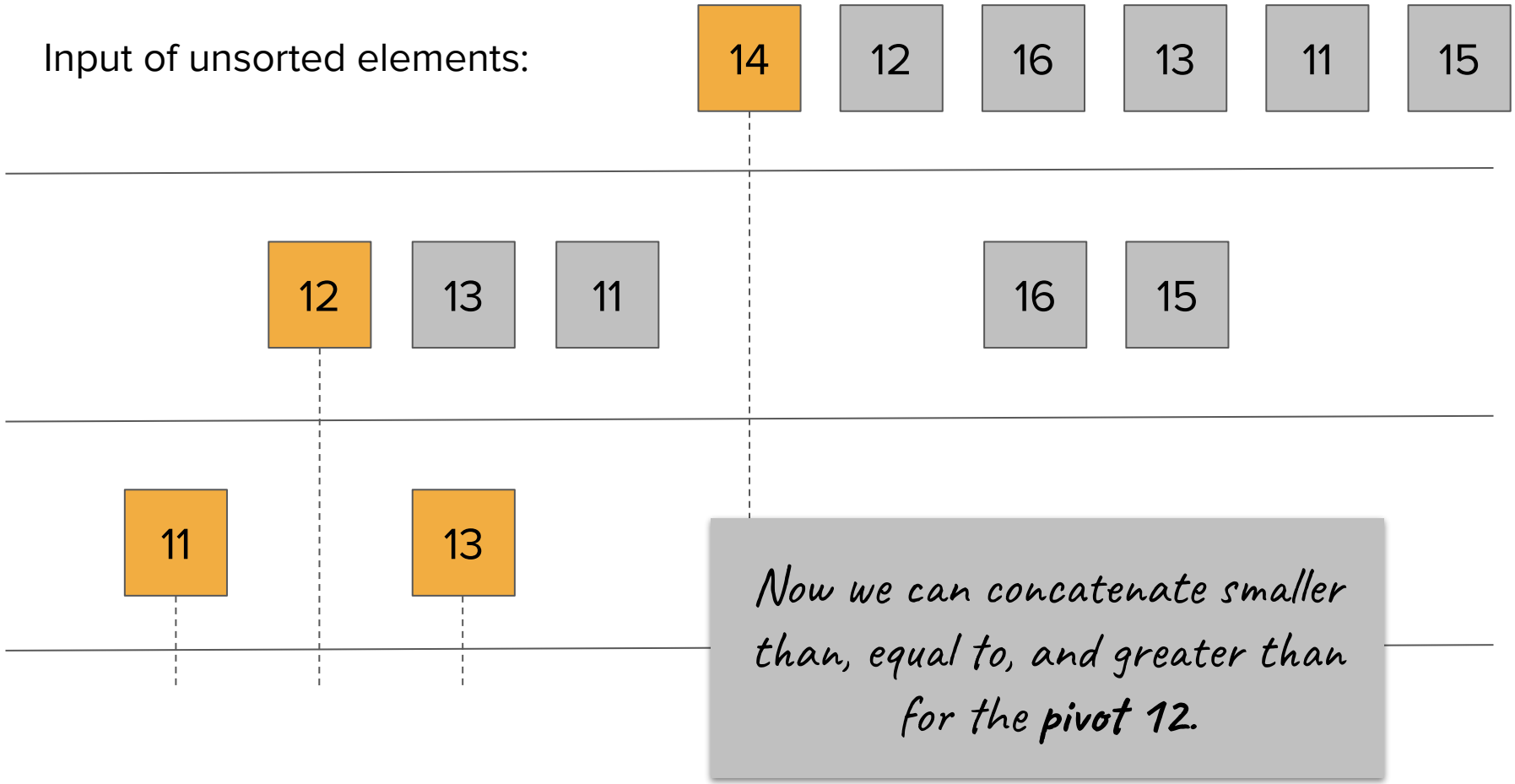

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13

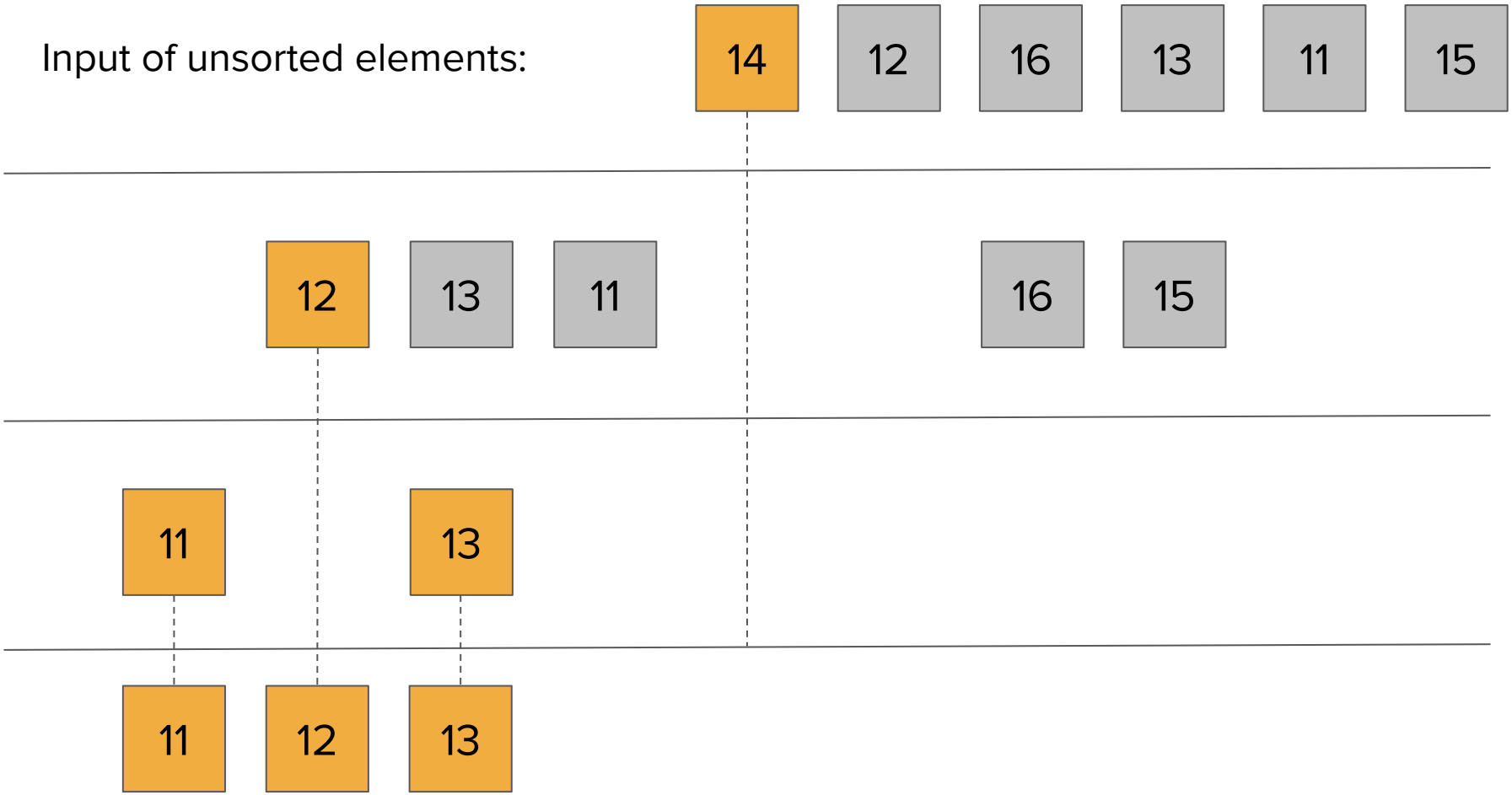

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
Recursively sort the smaller partition for pivot 12!

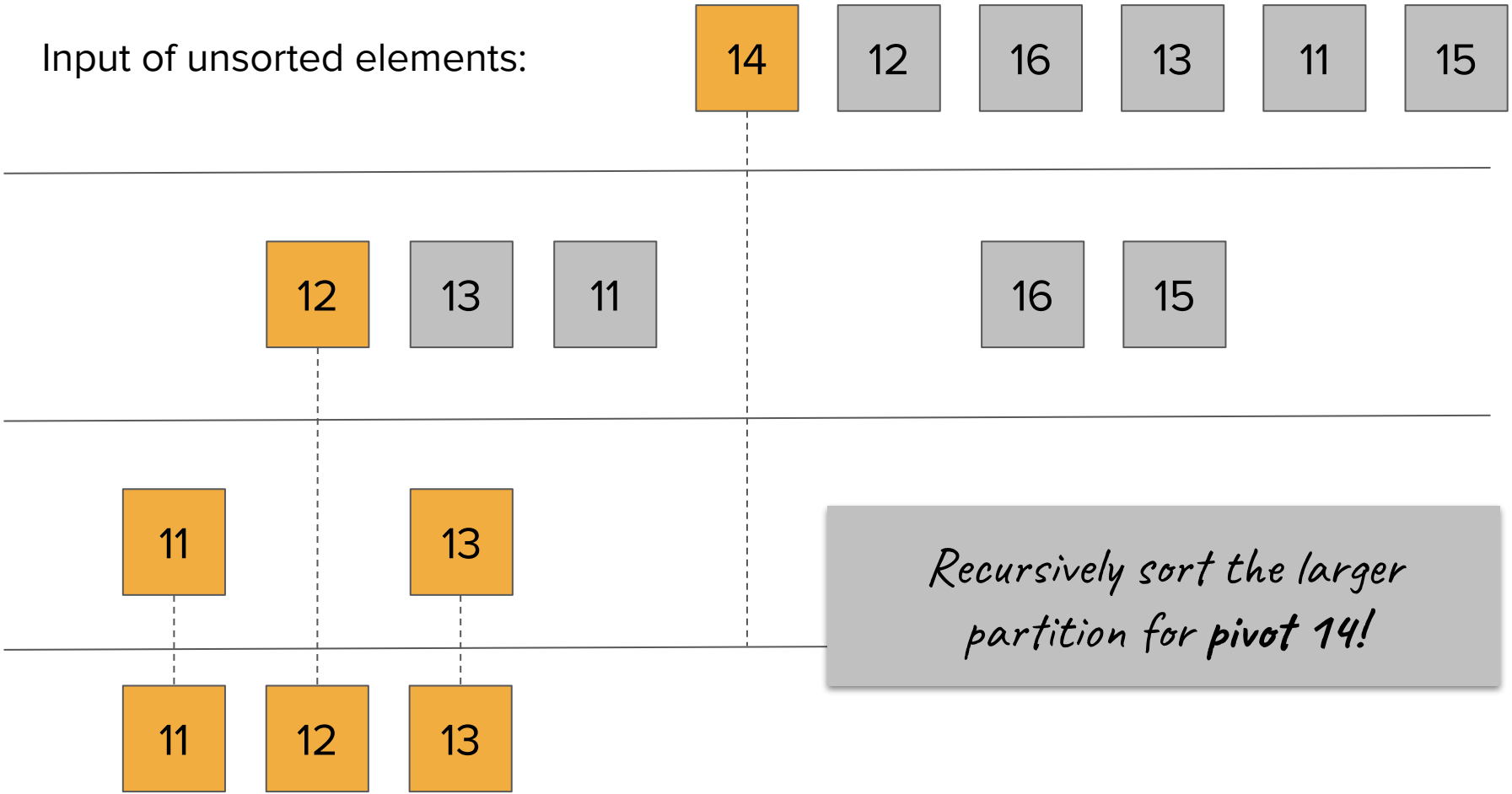

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
Only one element so we're done!

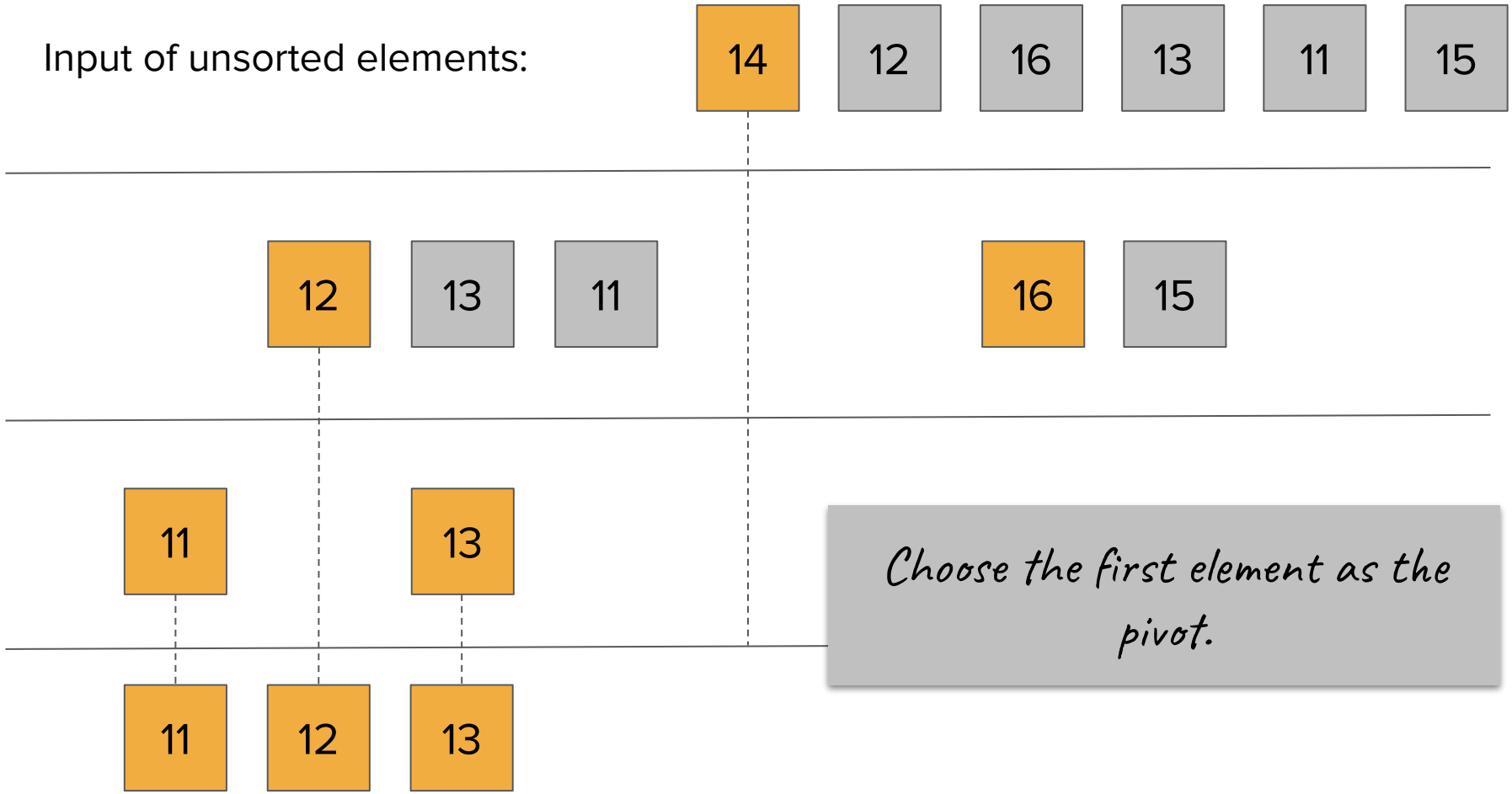

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
Recursively sort the larger partition for pivot 12!

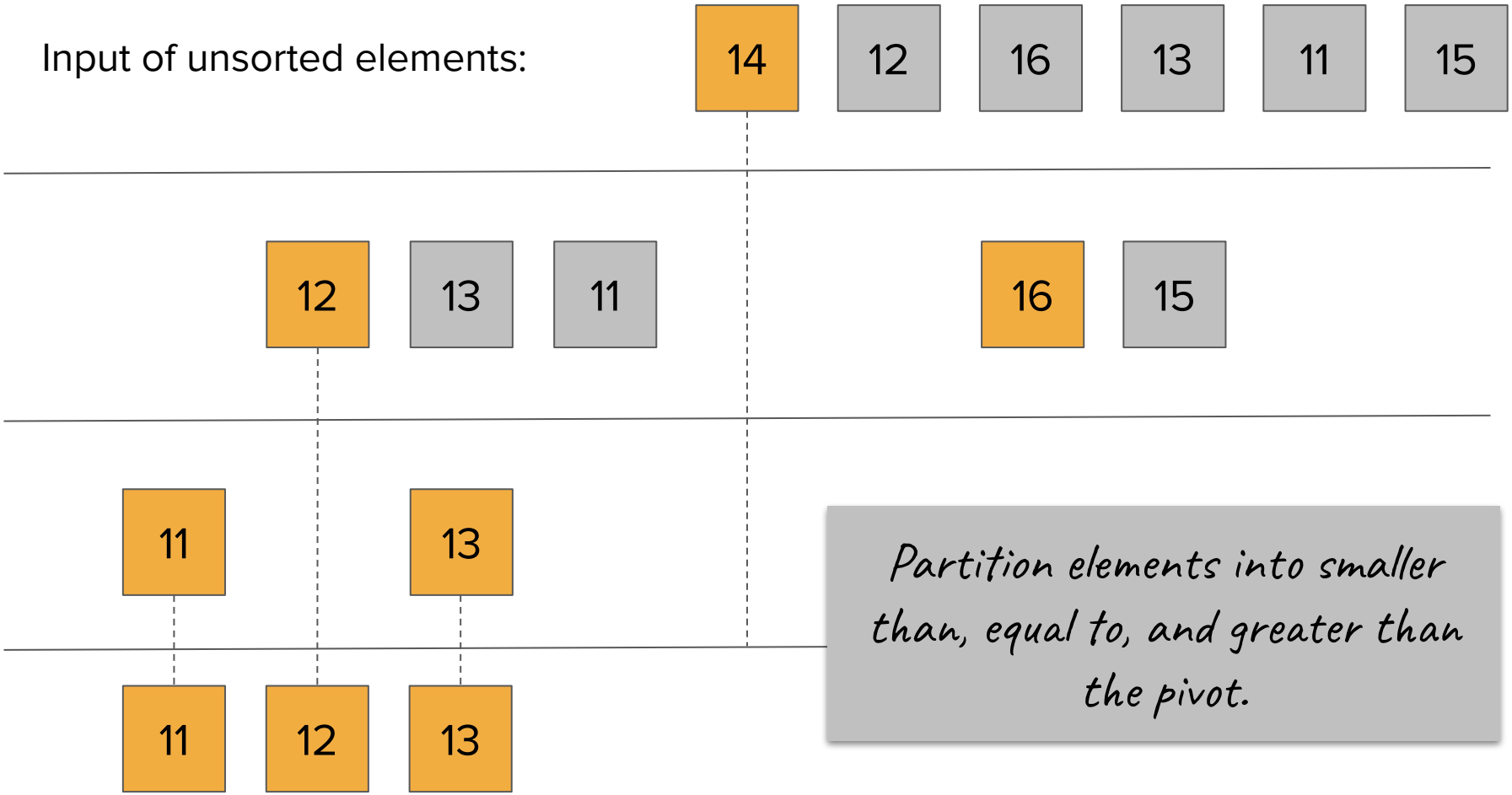

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
Only one element so we're done!

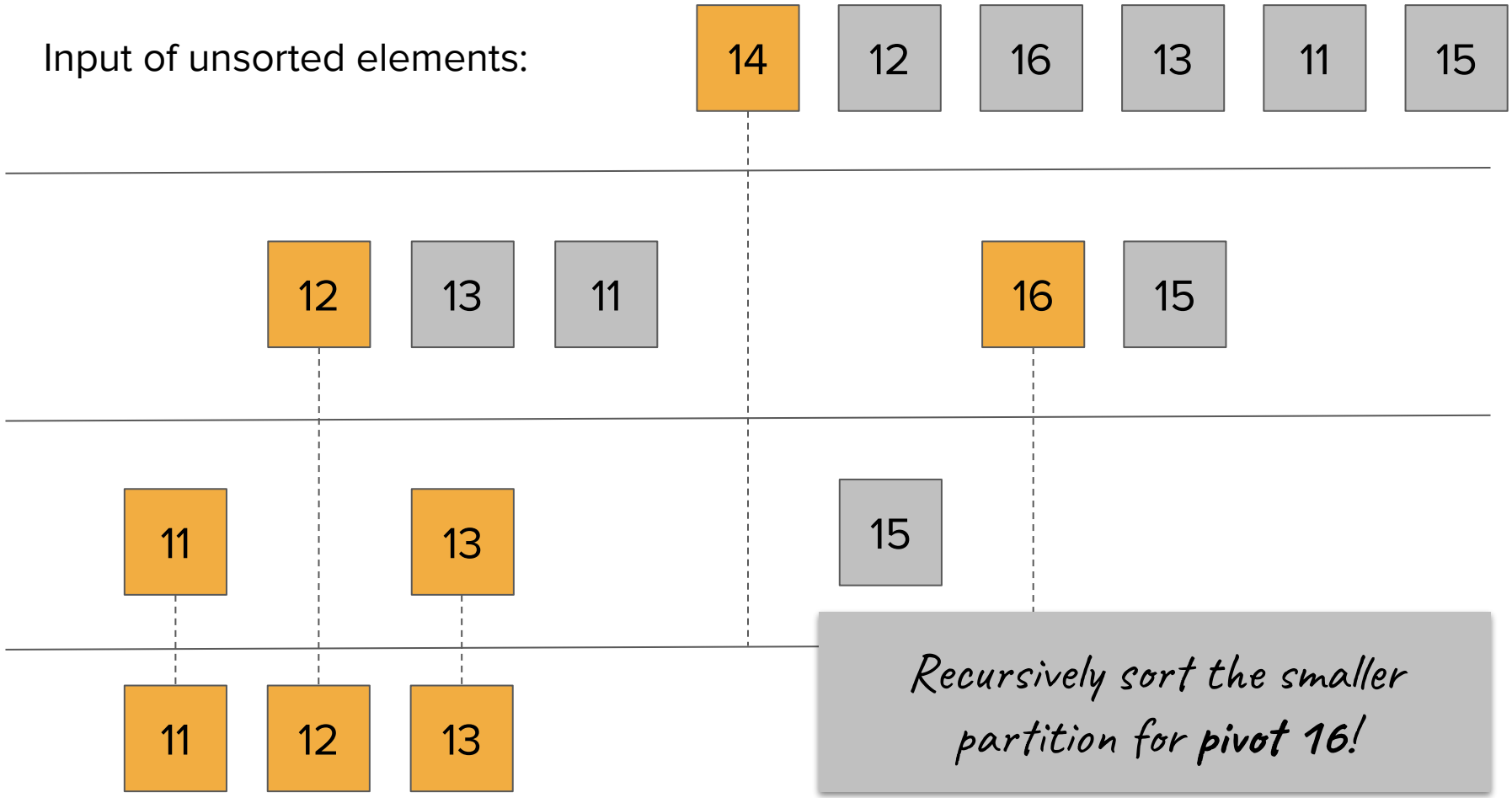

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
Now we can concatenate smaller than, equal to, and greater than for the pivot 12.

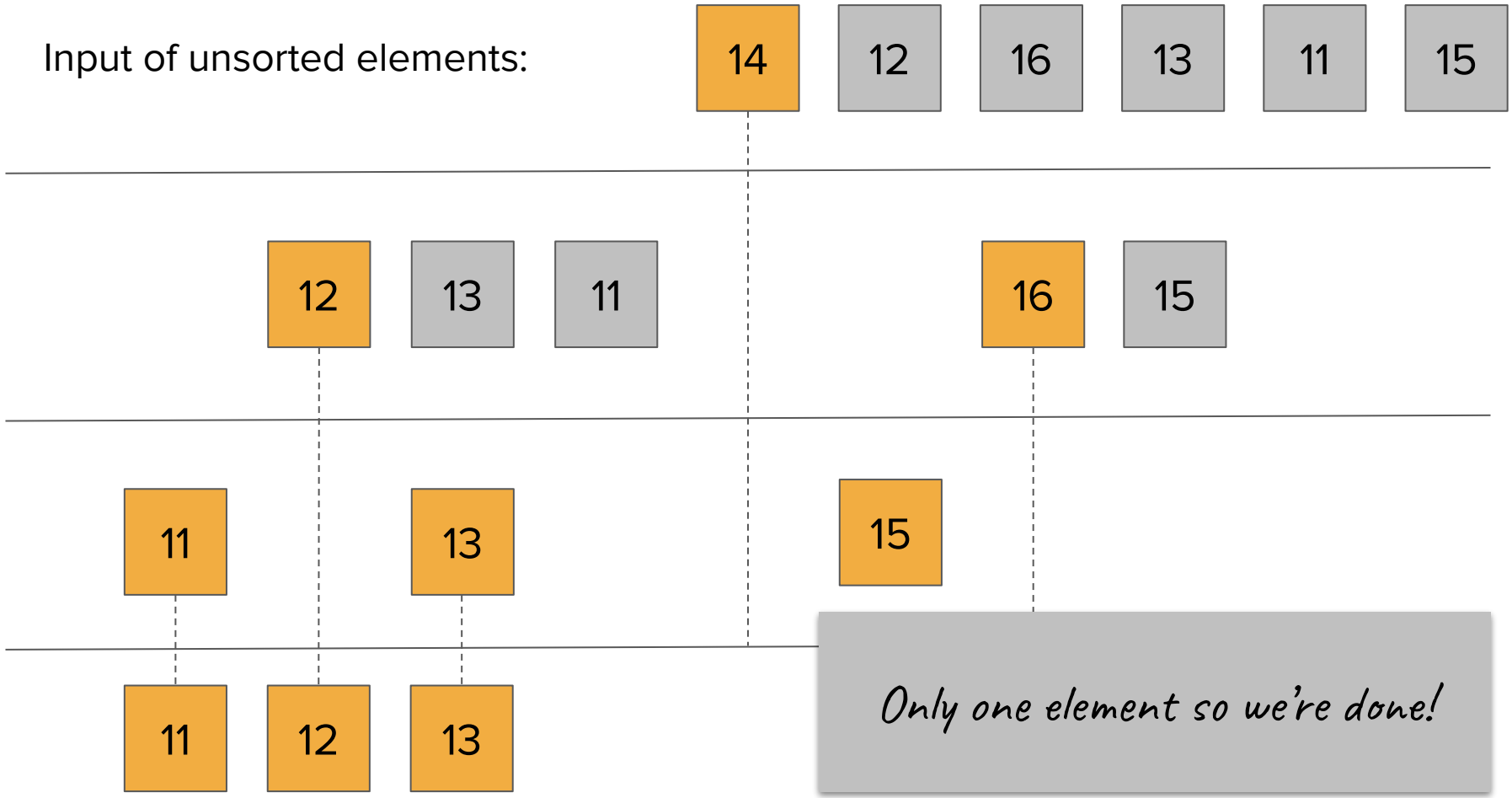

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
11
12
13

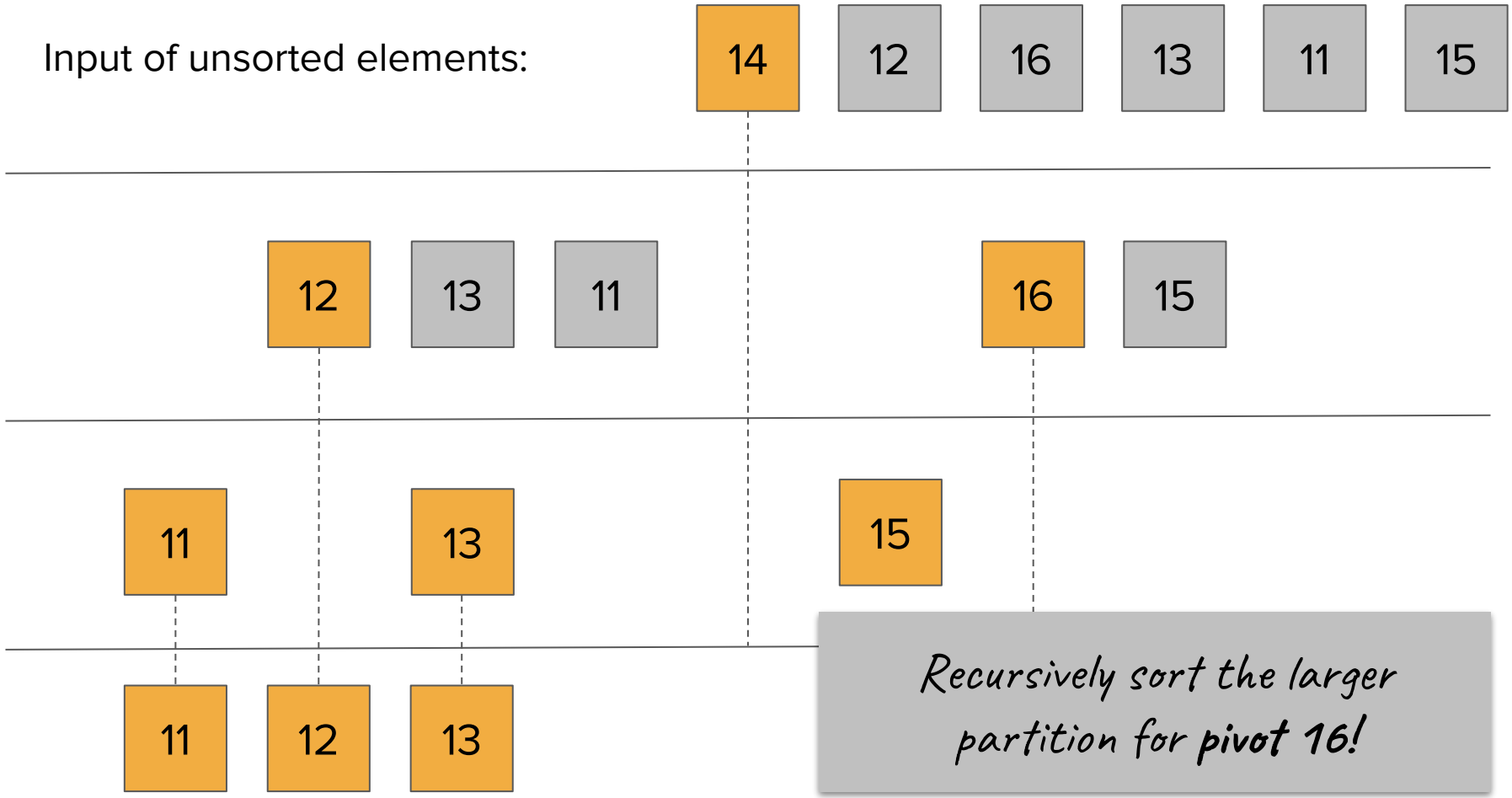

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
11
12
13
Recursively sort the larger partition for pivot 14!

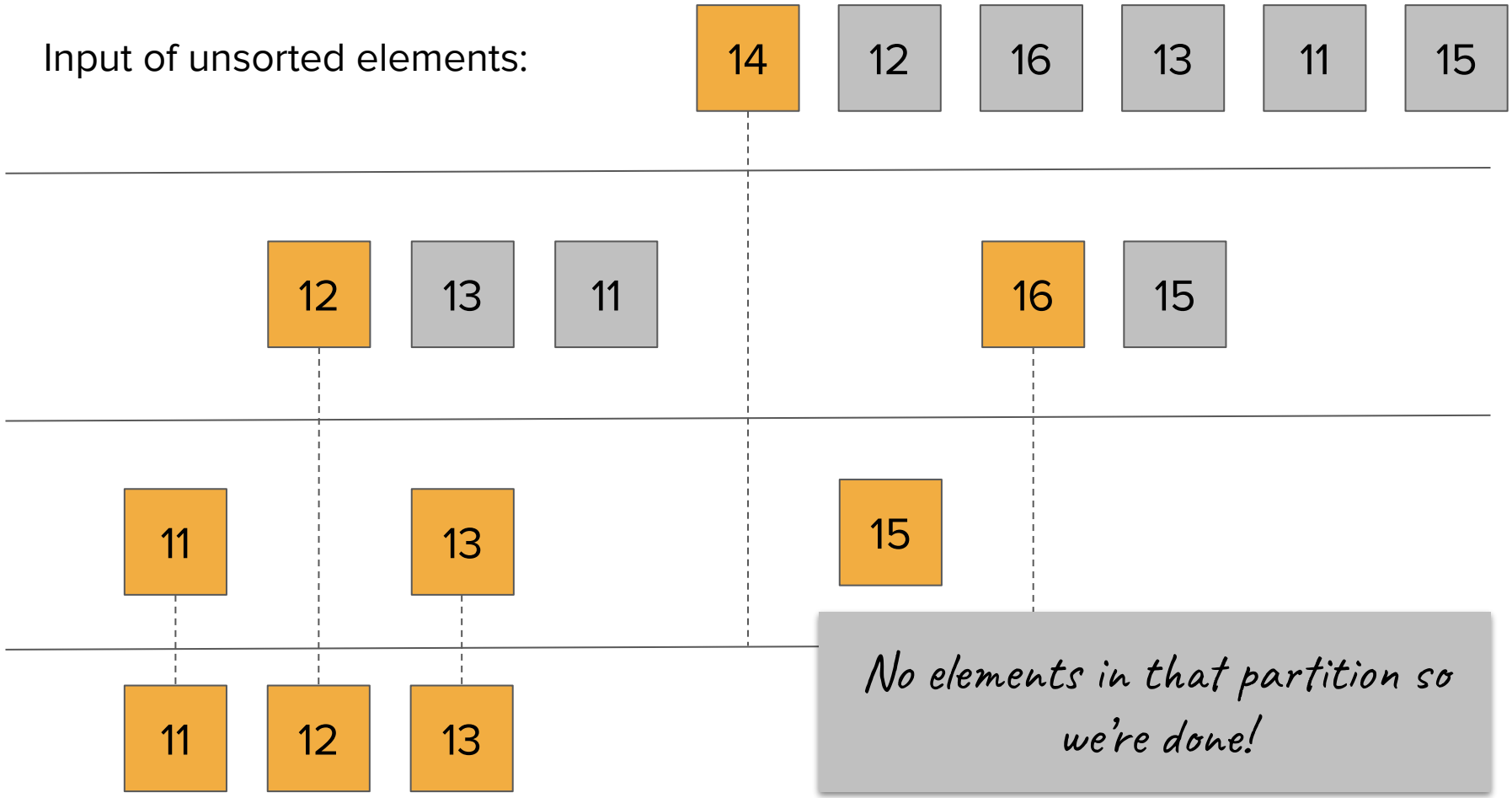

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
11
12
13
Choose the first element as the pivot.

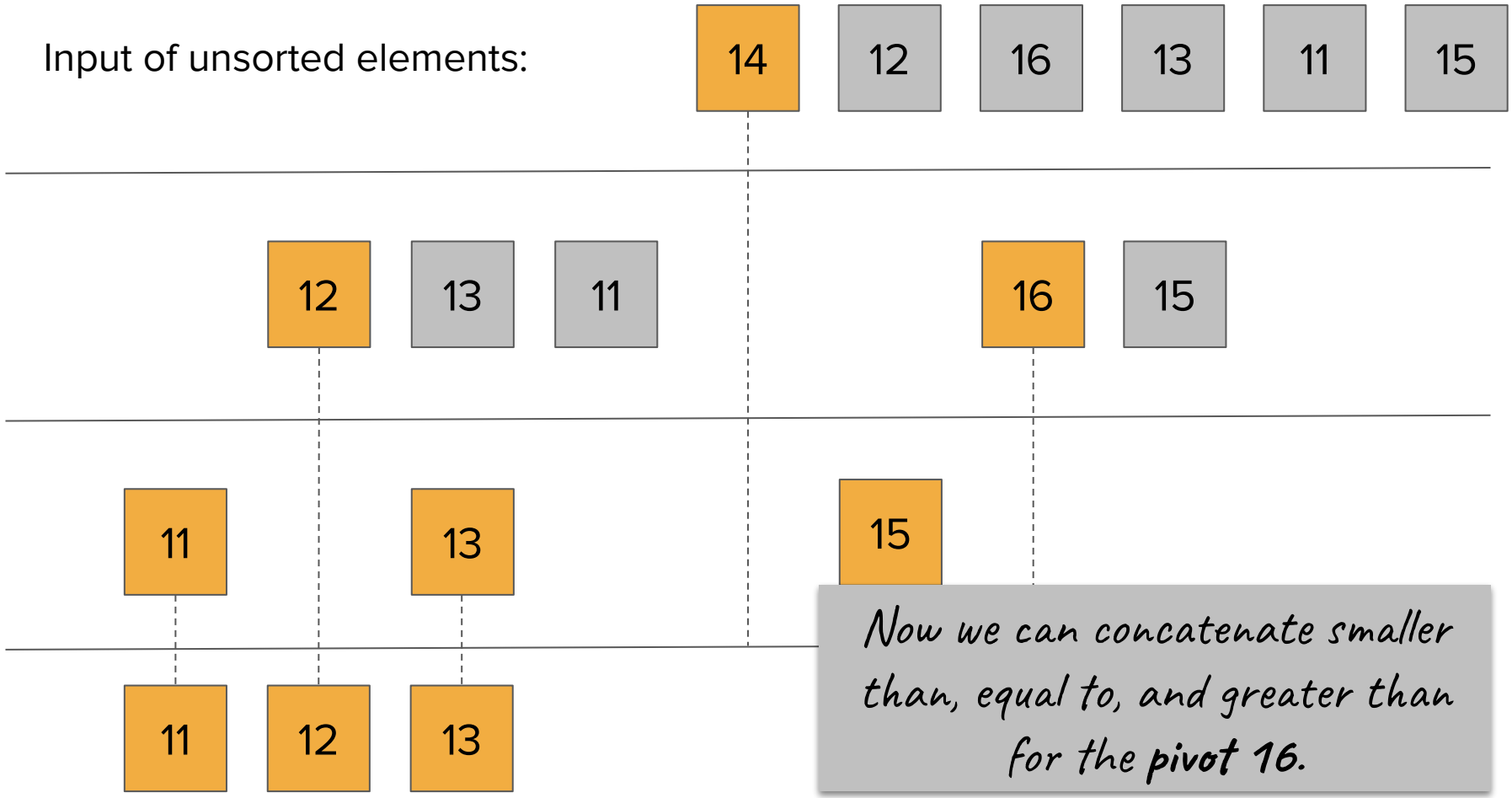

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
11
12
13
Partition elements into smaller than, equal to, and greater than the pivot.

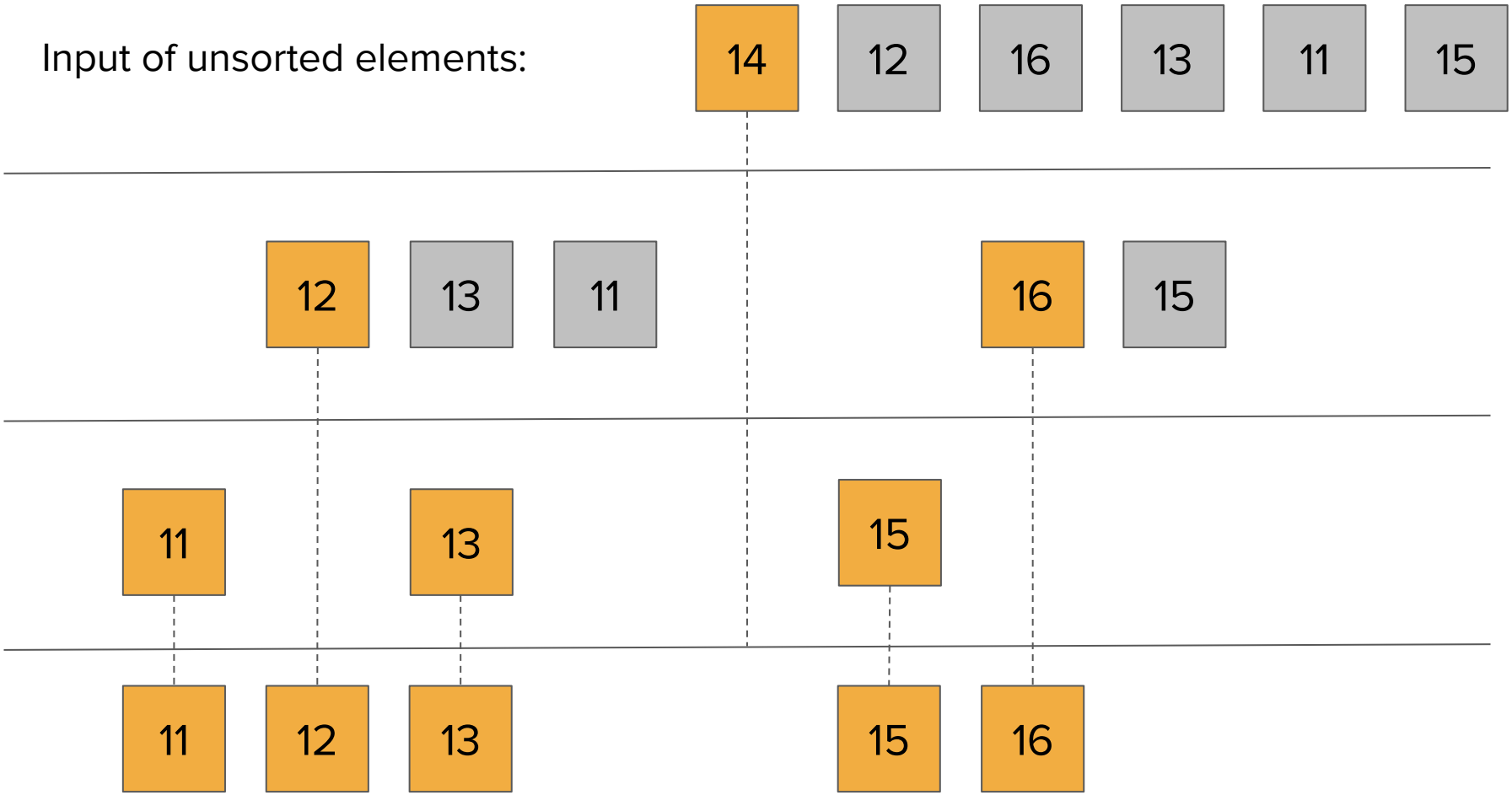

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
Recursively sort the smaller partition for pivot 16!

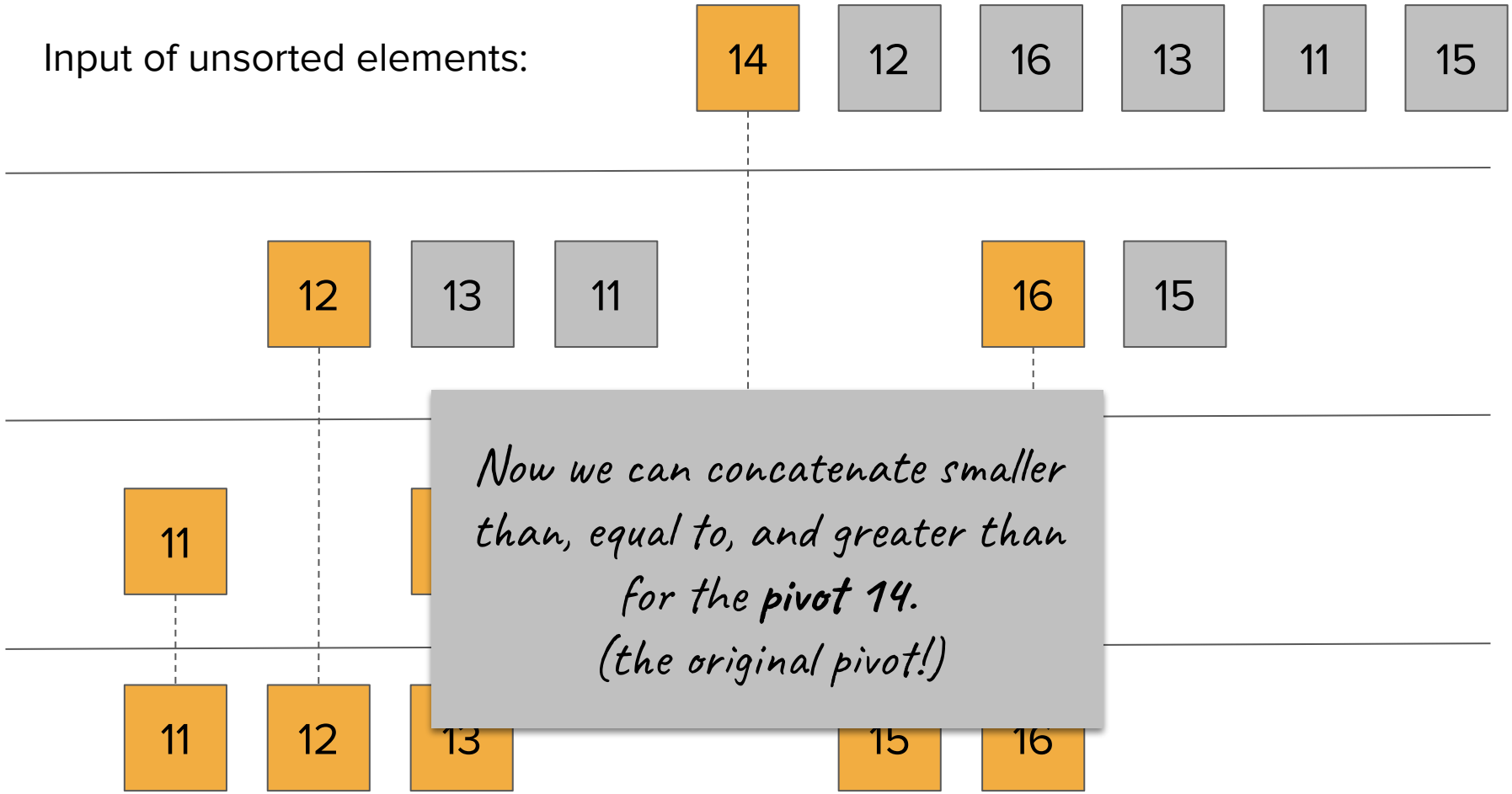

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
Only one element so we're done!

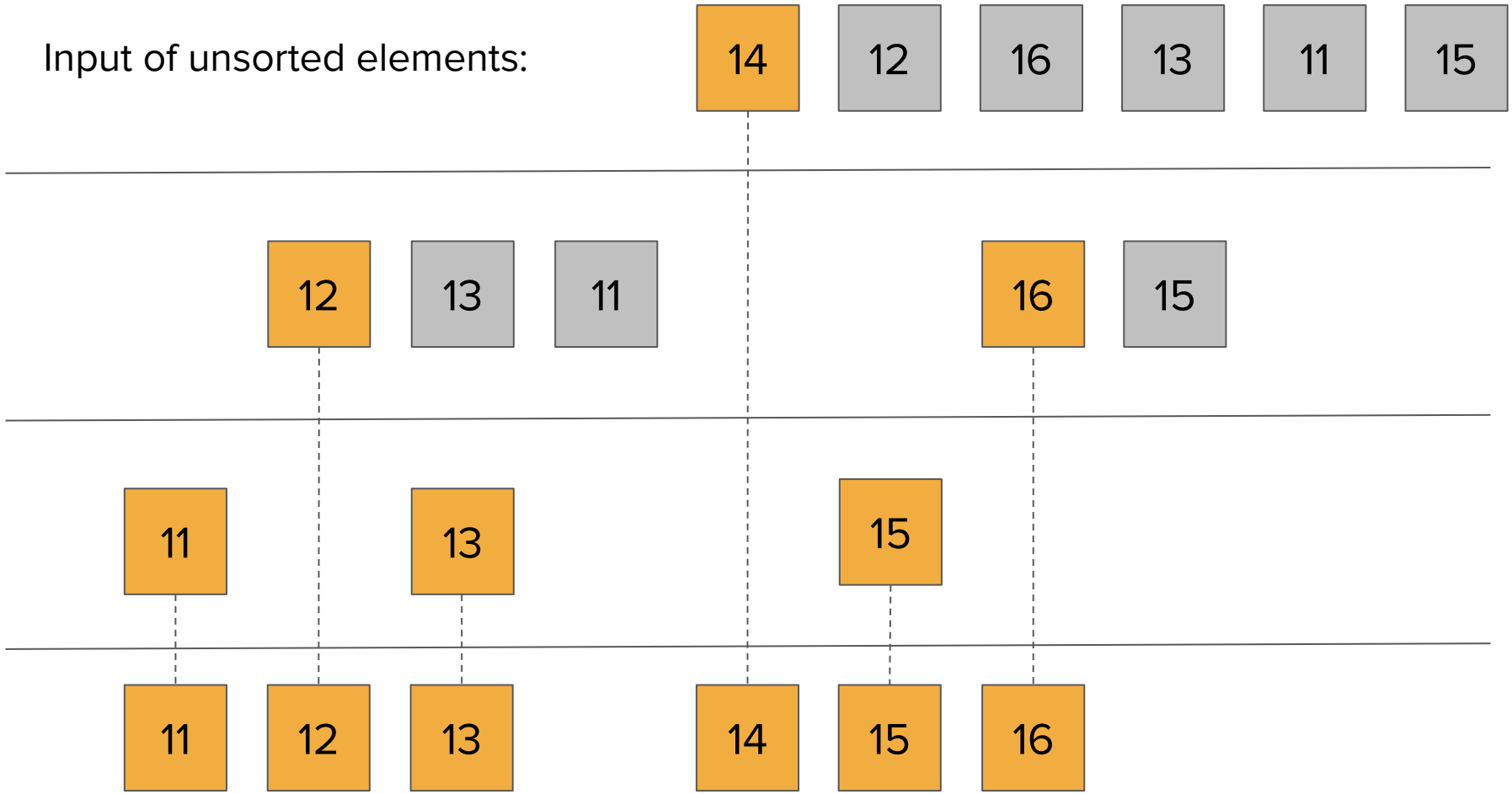

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
Recursively sort the larger partition for pivot 16!

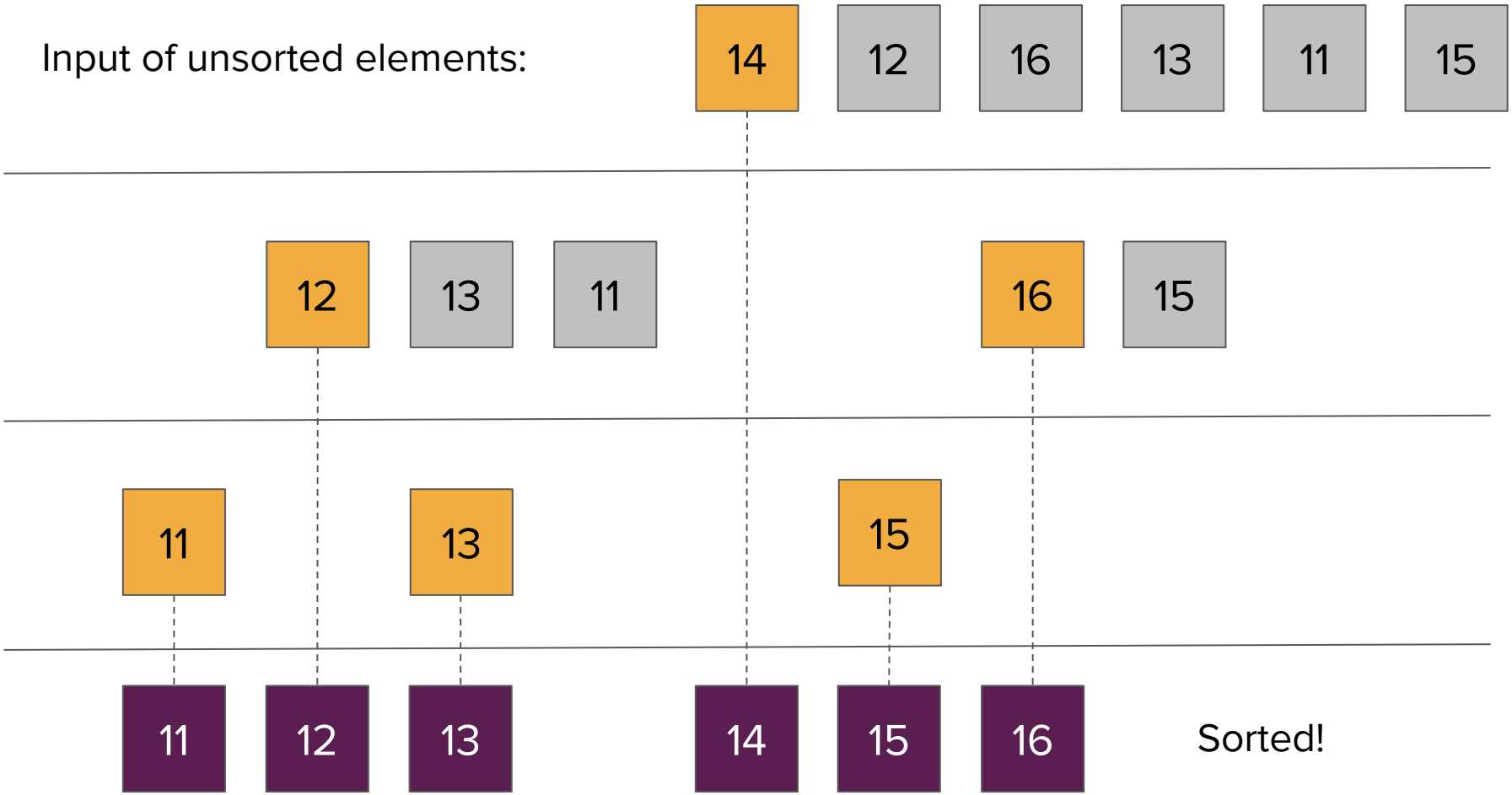

Input of unsorted elements:
14
12
16
13
11
15
12
13
11
16
15
11
13
15
11
12
13
No elements in that partition so we're done!


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 Pseudocode

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

    /* Pick the pivot and partition the list into three components.
     * 1) elements less than the pivot
     * 2) elements equal to the pivot
     * 3) elements greater than the pivot
    */
    Define three empty lists: less, equal, greater
    pivot = first element of vector (arbitrary choice)
    partition(vec, less, equal, greater, pivot)

    /* Recursively sort the two unsorted components. */
    quickSort(less)
    quickSort(greater)

    /* Concatenate newly sorted results and store in original vector */
    concatenate(vec, less, equal, greater)
}
```

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**.
  - Take CS161 to learn how to write this proof!

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**.
  - Take CS161 to learn how to write this proof!

- Thus, we can't do better (in Big-O terms at least) than Merge Sort and Quicksort!
  - Take CS161 to learn about how there are actually clever non-comparison-based sorting algorithms that are able to break this limit.

Final Advice

Assignment 6 Tips

- When implementing the sorting algorithm on linked lists, it is strongly recommended to implement helper functions for the divide/join components of the algorithm.
  - For quicksort this means having helper functions for the partition and concatenate operations

Assignment 6 Tips

- When implementing the sorting algorithm on linked lists, it is strongly recommended to implement helper functions for the divide/join components of the algorithm.
  - For quicksort this means having helper functions for the partition and concatenate operations

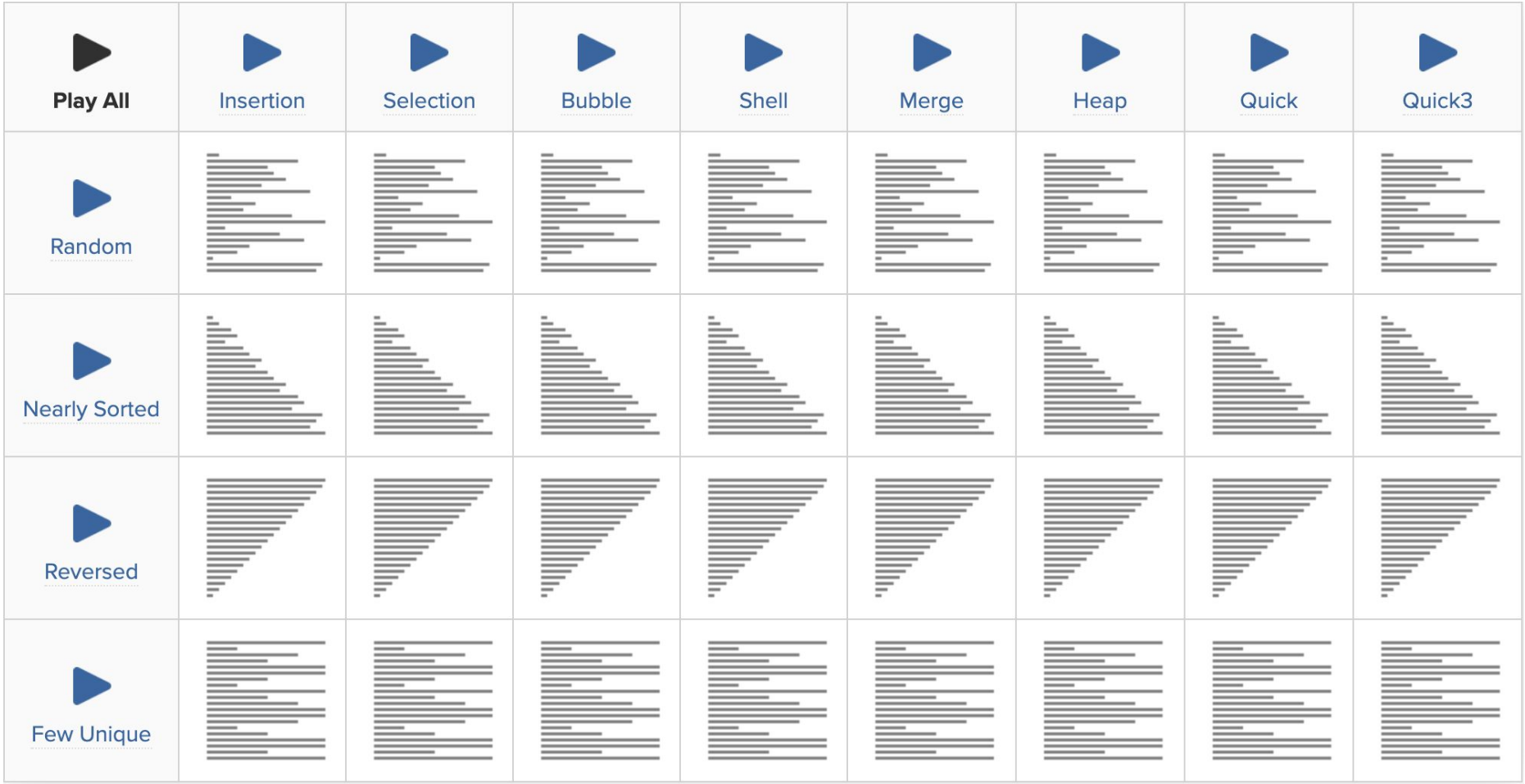
- These helper functions should be implemented iteratively, but the overall sorting algorithms itself operates recursively. Mind the distinction!

Assignment 6 Tips

- When implementing the sorting algorithm on linked lists, it is strongly recommended to implement helper functions for the divide/join components of the algorithm.
  - 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 itself operates recursively. Mind the distinction!
- Write tests for your helper functions first! Then, write end-to-end tests for your sorting function.

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!

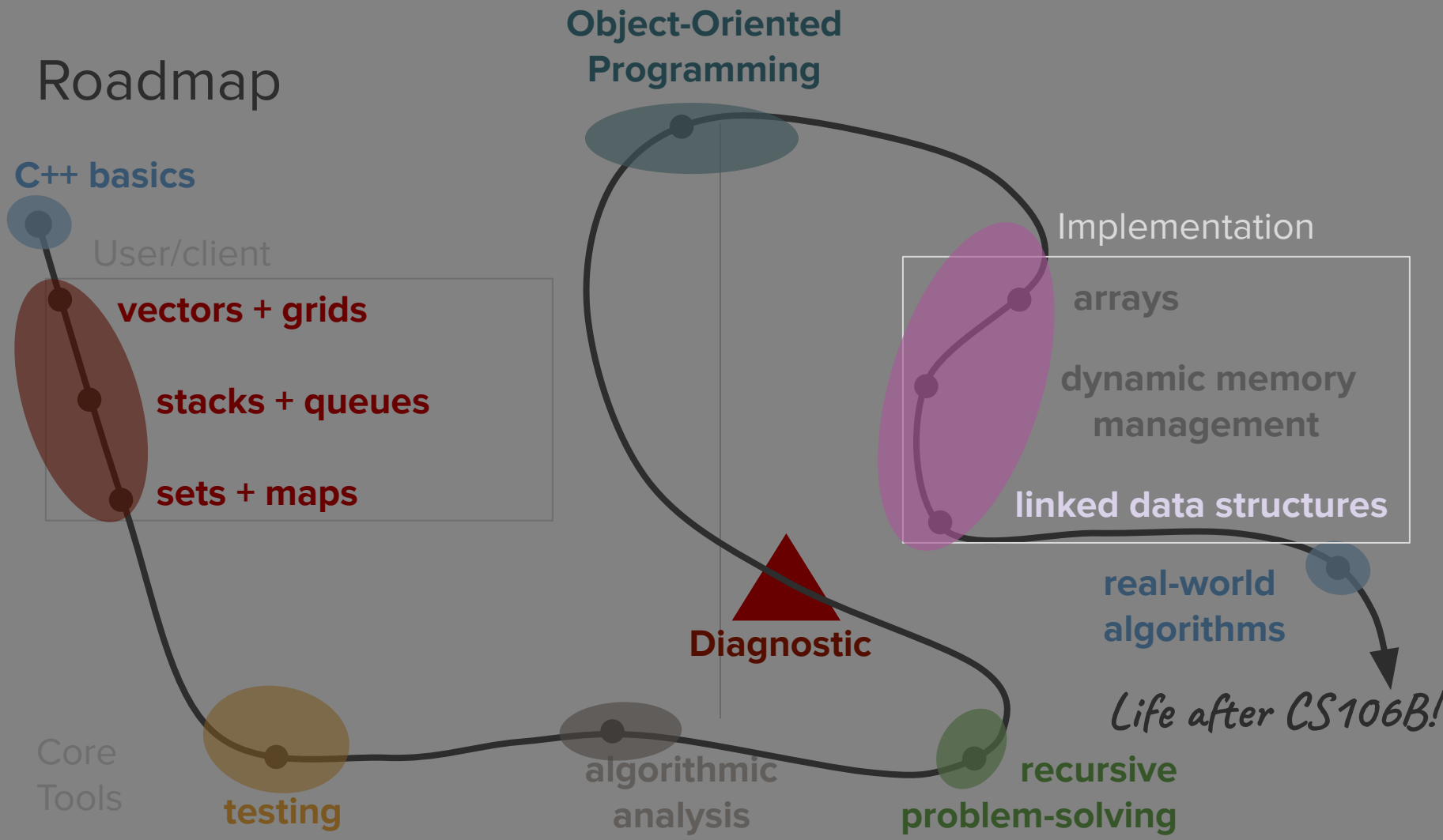
- There are many different methods for sorting data:
  - Selection Sort
  - Insertion Sort
  - Mergesort
  - Quicksort
  - And many more...

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


std::stable_sort (gcc) - 8950 comparisons, 20268 array accesses, 1.00 ms delay
http://panthema.net/2013/sound-of-sorting

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!