

Sorting

Tiziana Ligorio

Today's Plan



Sorting algorithms and
their analysis

Sorting

Rearranging a sequence into increasing
(decreasing) order!

Several approaches

Can do it in many ways

What is the best way?

Let's find out using Big-O

Lecture Activity

Write **pseudocode** to sort an array.

543	3	523	76	200	158	195	108	43	274	100	14	599
-----	---	-----	----	-----	-----	-----	-----	----	-----	-----	----	-----

There are many approaches to sorting
We will look at some comparison-
based approaches here

Selection Sort

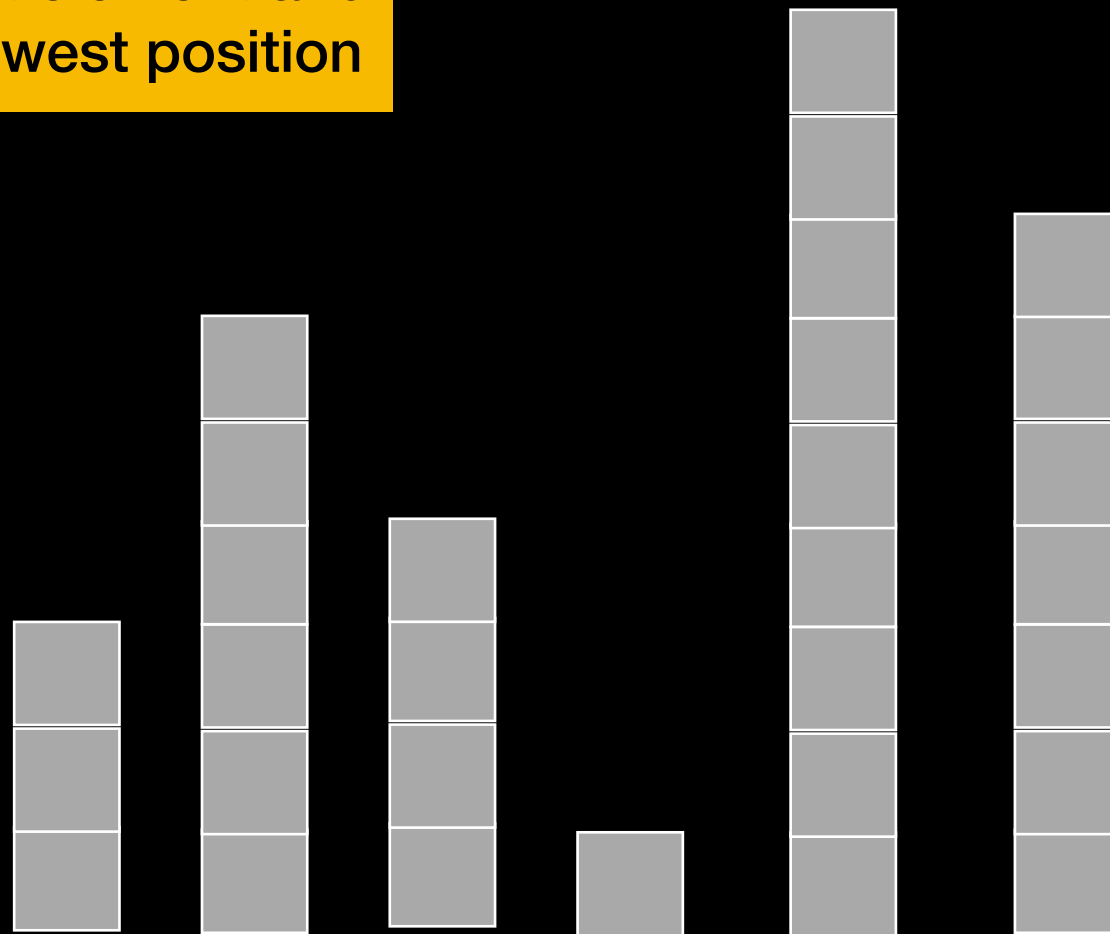
Selection Sort



Find smallest element and
move it at lowest position



1st Pass



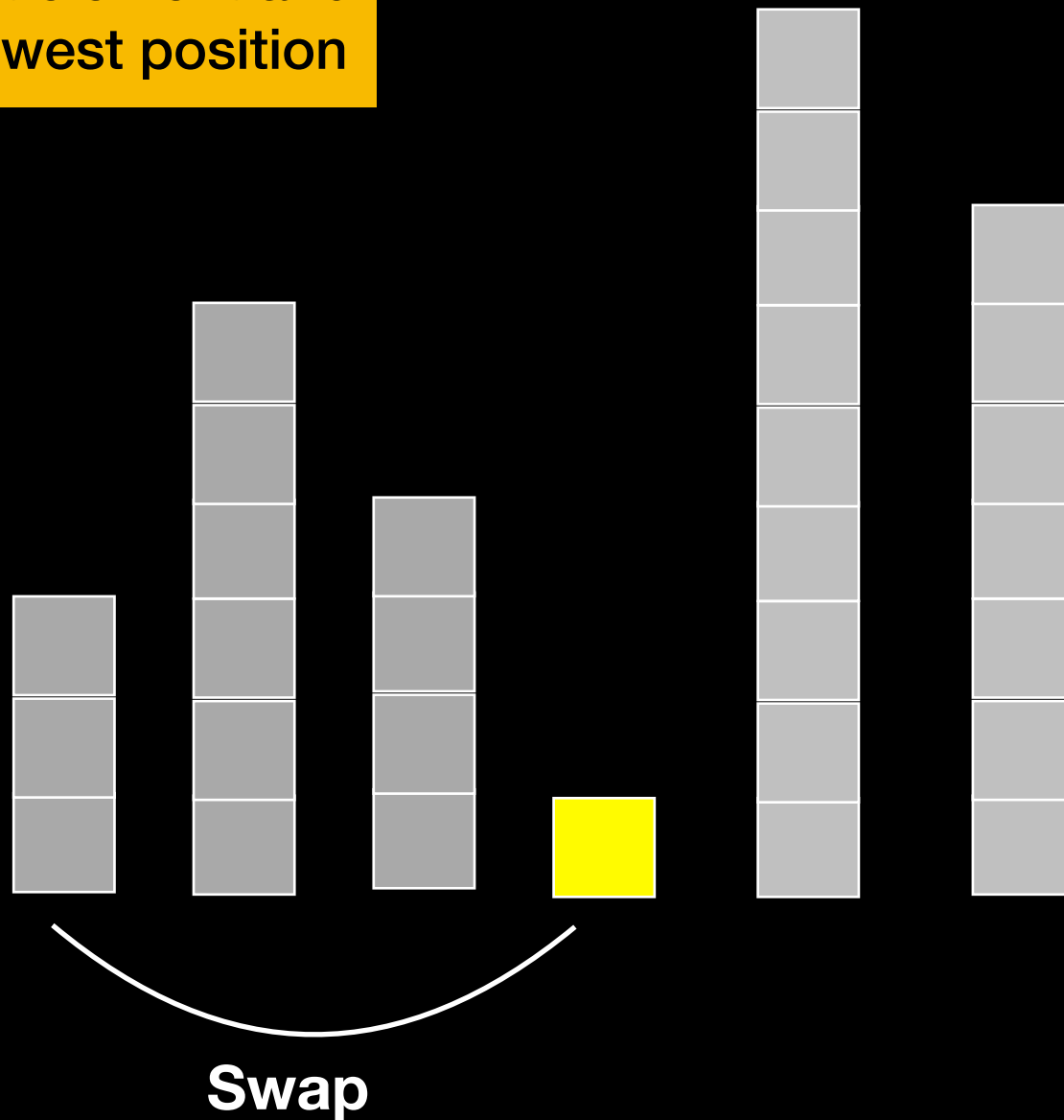
Selection Sort



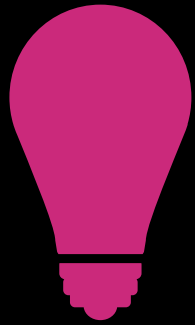
Find smallest element and
move it at lowest position




1st Pass



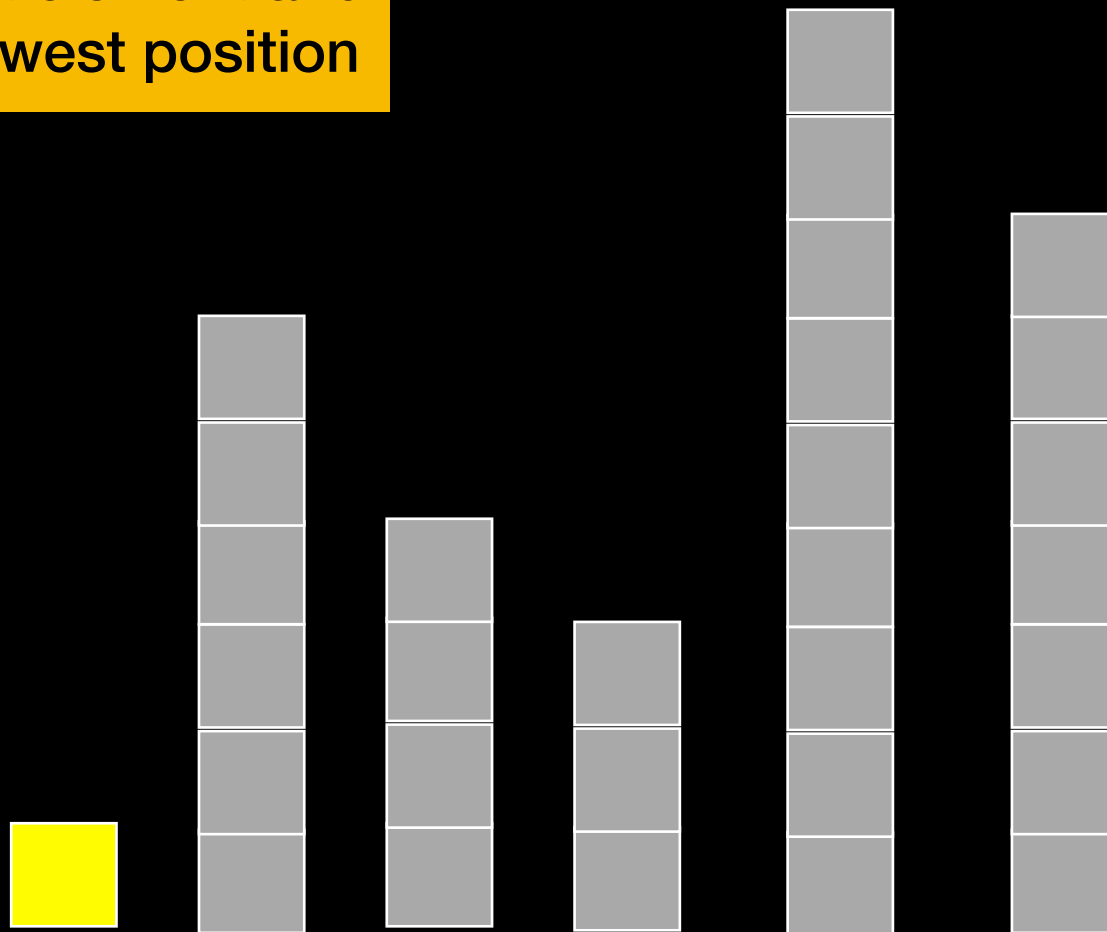
Selection Sort



Find smallest element and
move it at lowest position

 Unsorted
 Sorted

1st Pass



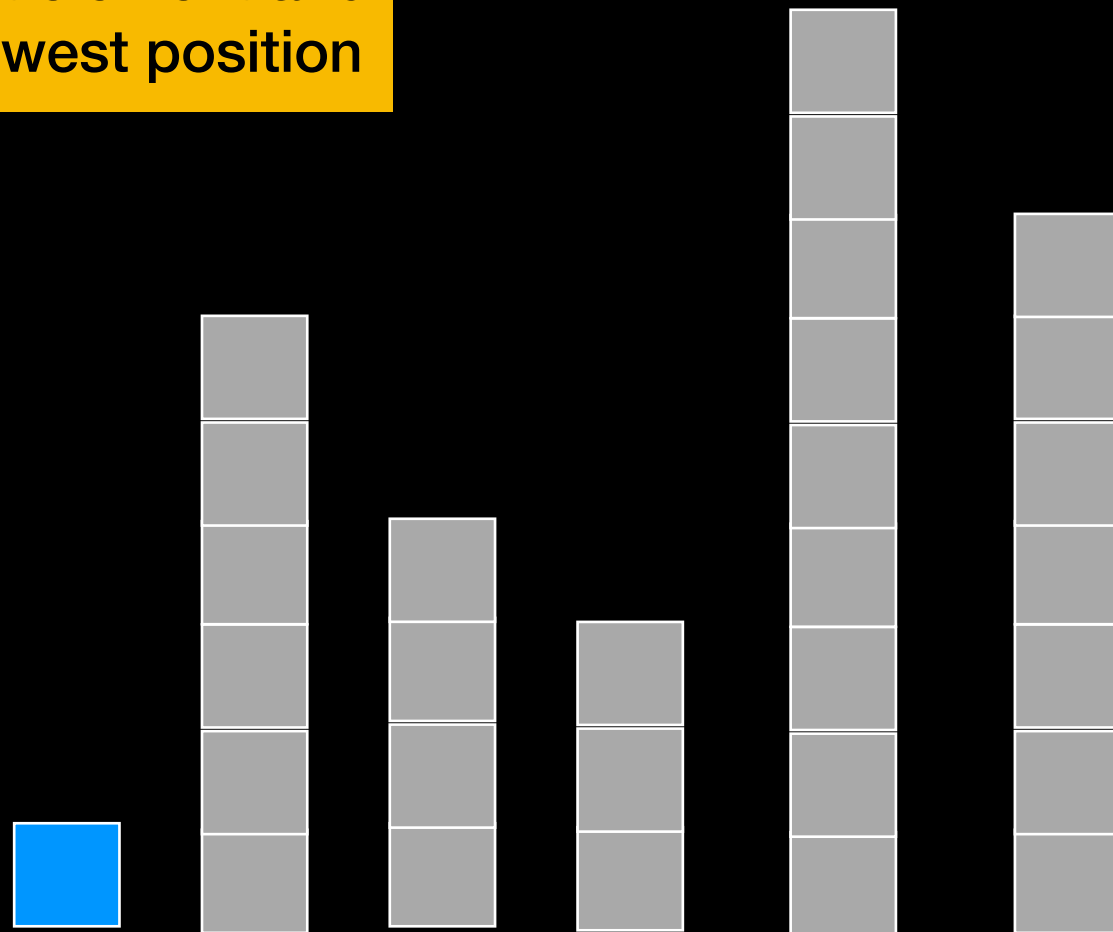
Selection Sort



Find smallest element and
move it at lowest position

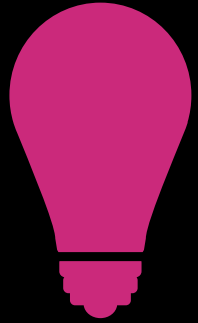


2nd Pass



Unsorted

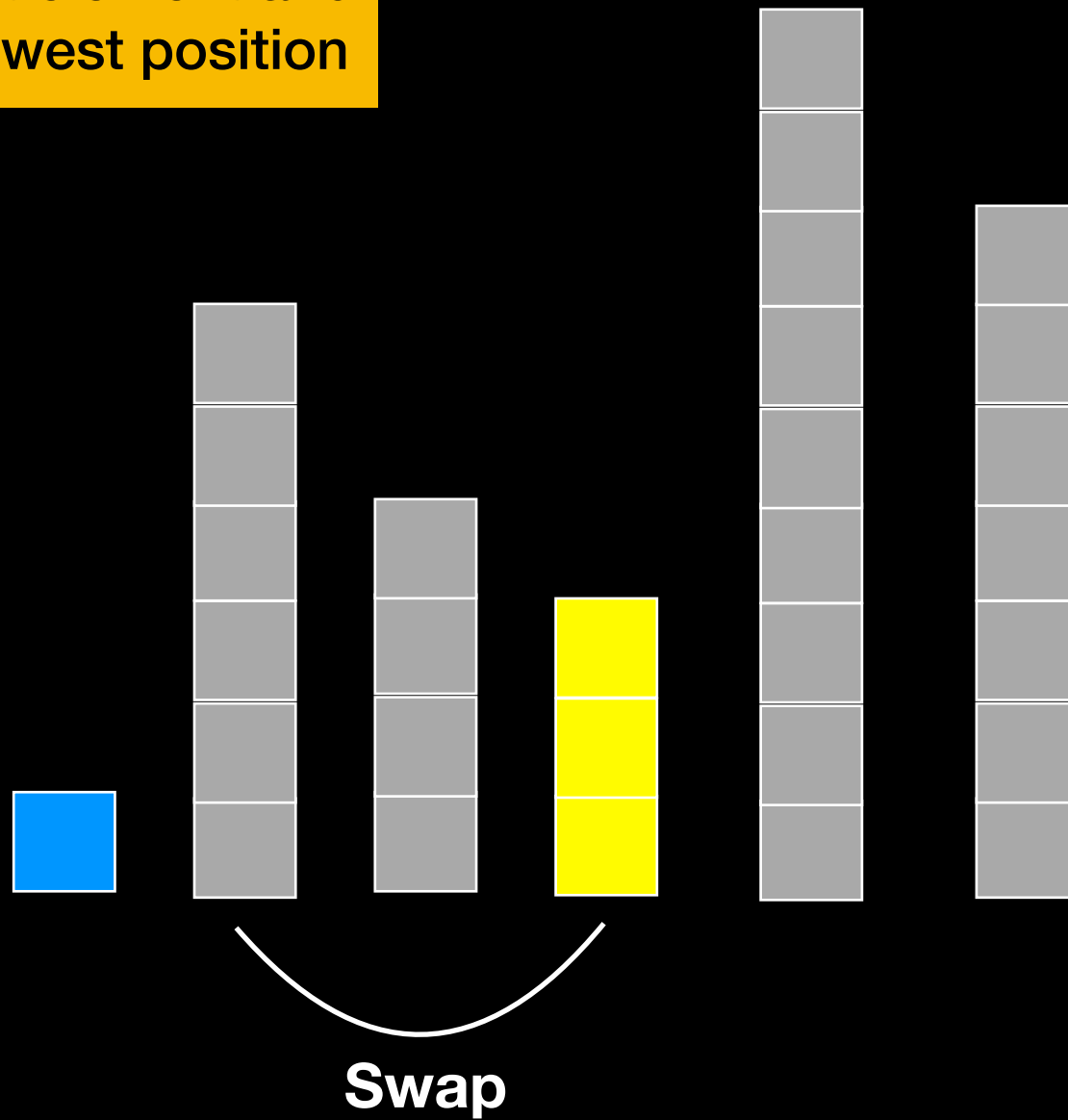
Selection Sort



Find smallest element and
move it at lowest position



2nd Pass



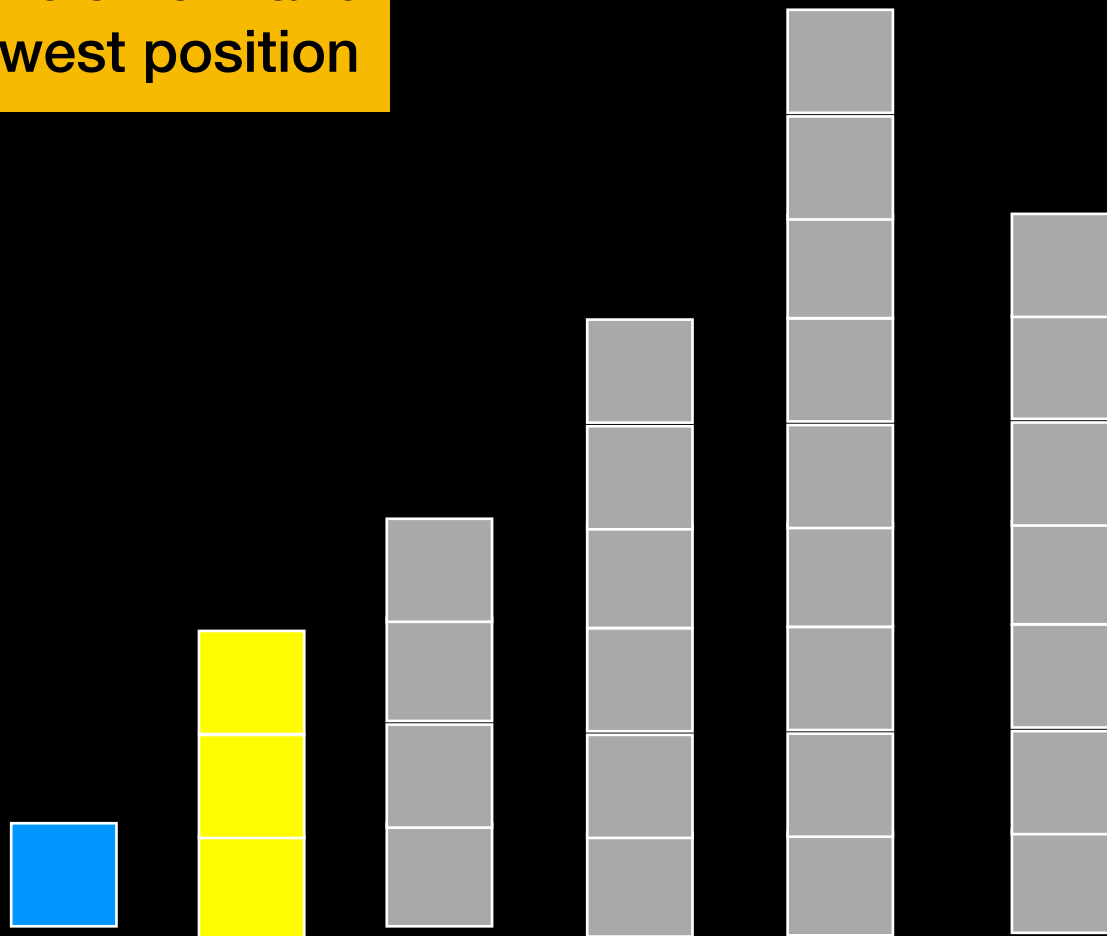
Selection Sort



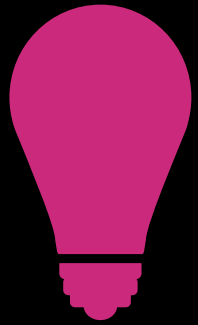
Find smallest element and
move it at lowest position



2nd Pass



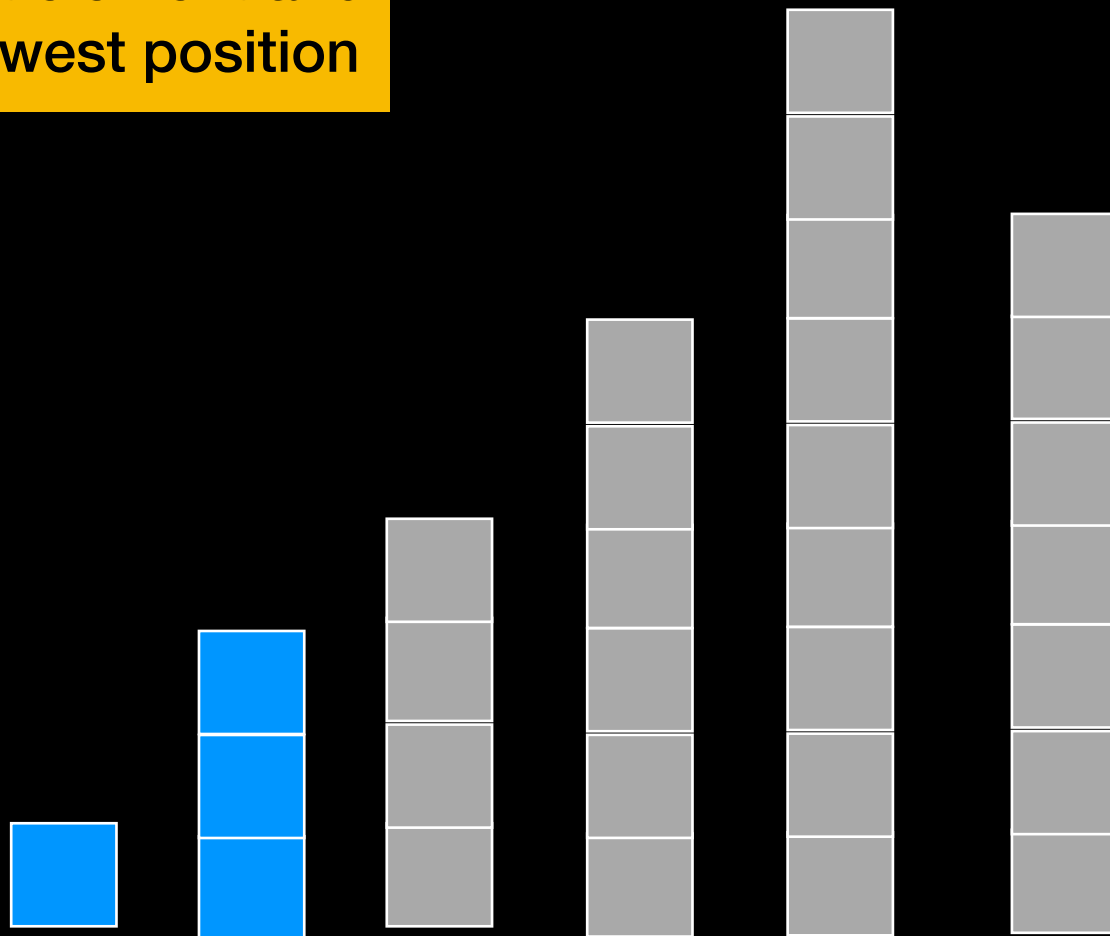
Selection Sort



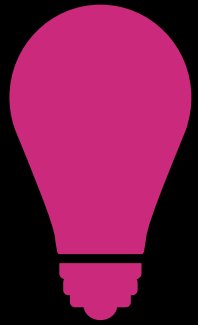
Find smallest element and
move it at lowest position

Unsorted
Sorted

3rd Pass



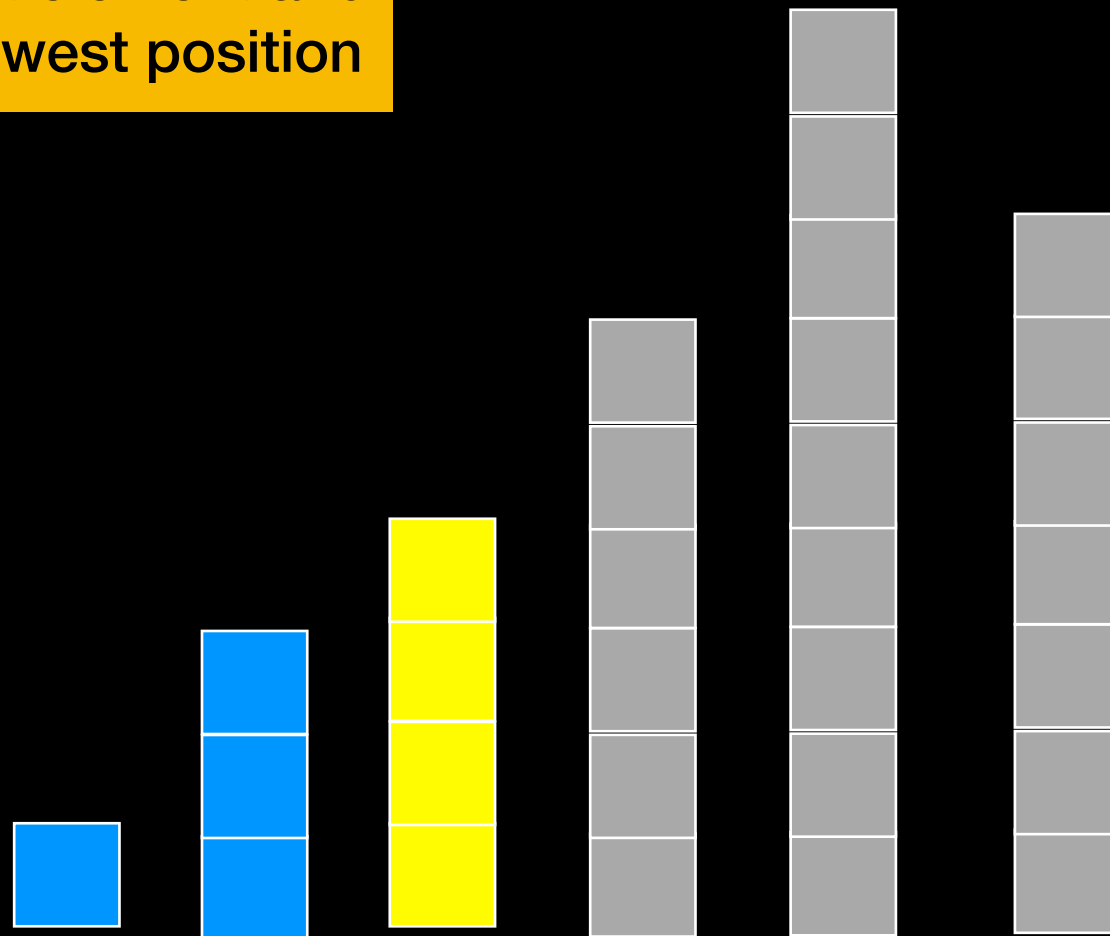
Selection Sort



Find smallest element and
move it at lowest position



3rd Pass



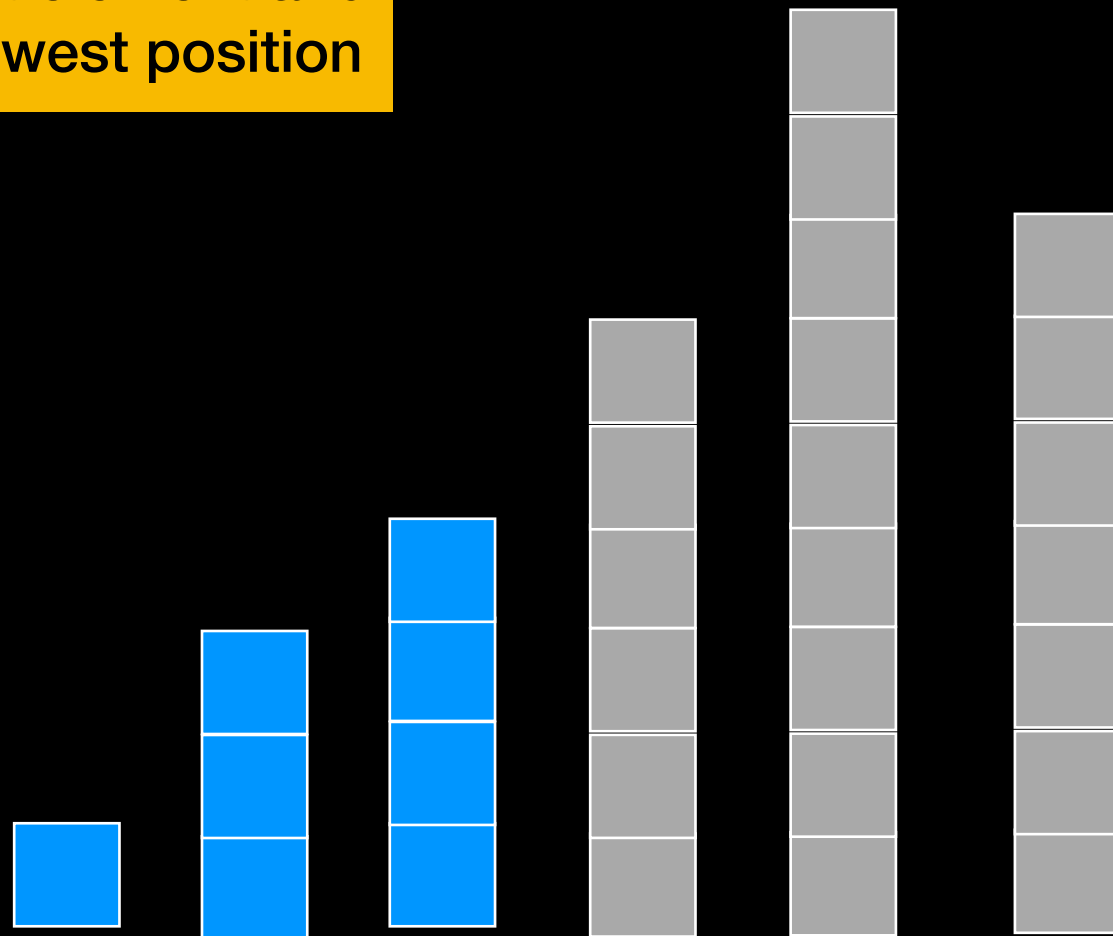
Selection Sort



Find smallest element and
move it at lowest position



4th Pass



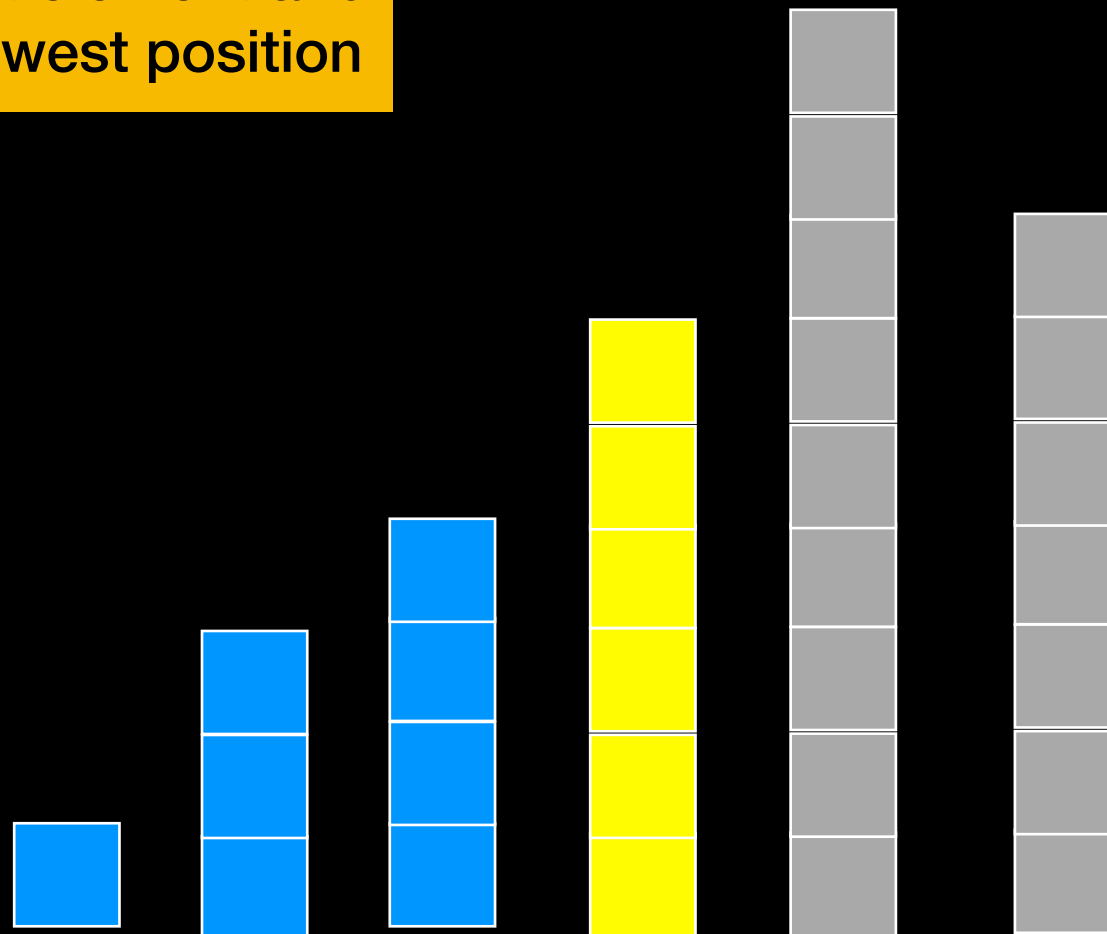
Selection Sort



Find smallest element and
move it at lowest position



4th Pass



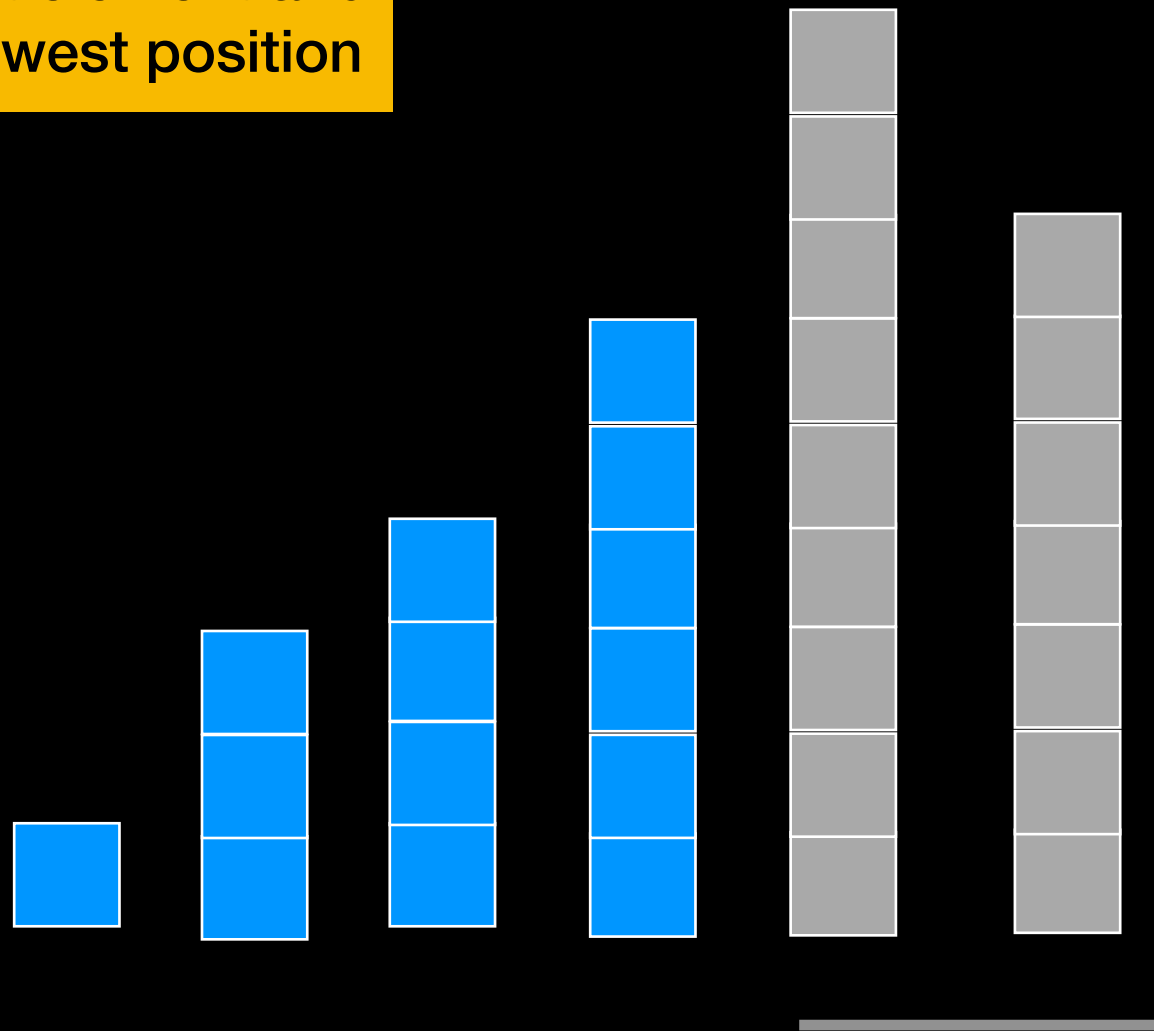
Selection Sort



Find smallest element and
move it at lowest position



5th Pass



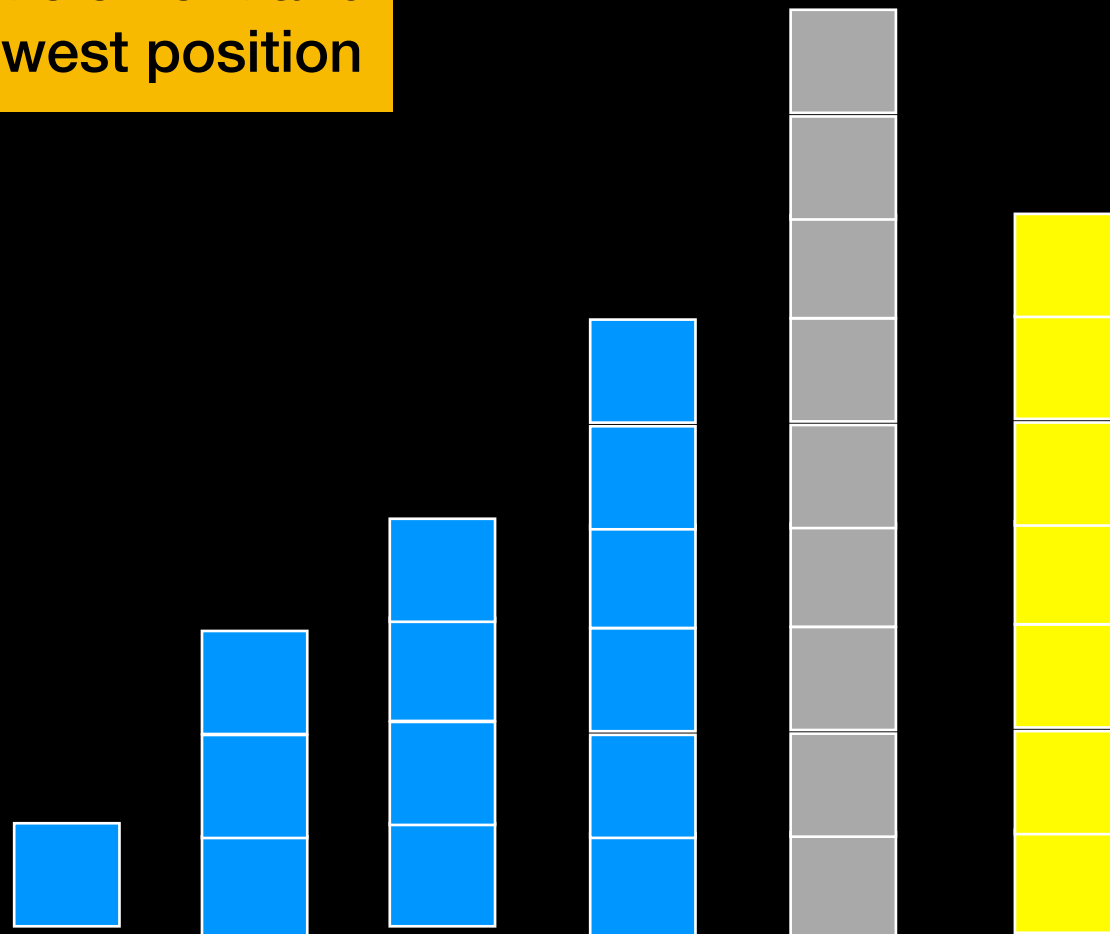
Selection Sort



Find smallest element and
move it at lowest position

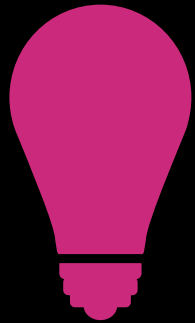


5th Pass



Swap

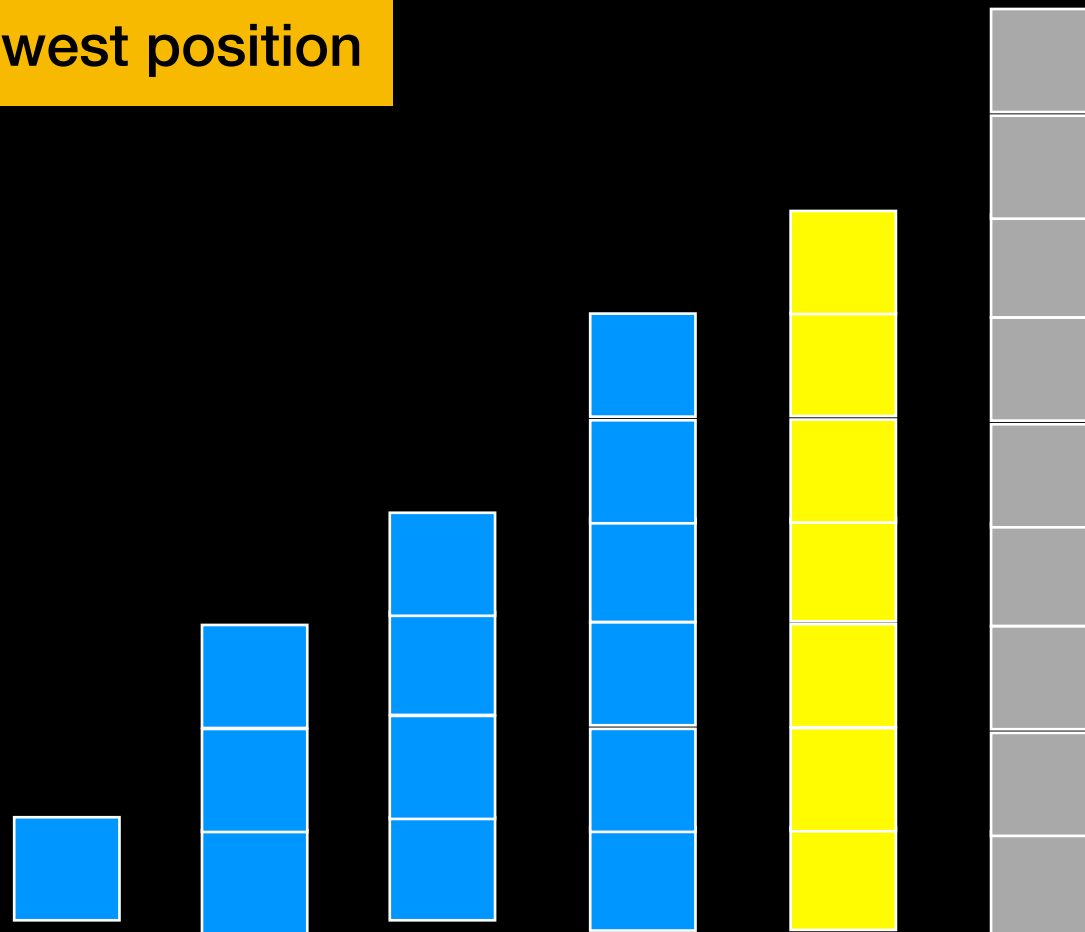
Selection Sort



Find smallest element and
move it at lowest position



5th Pass



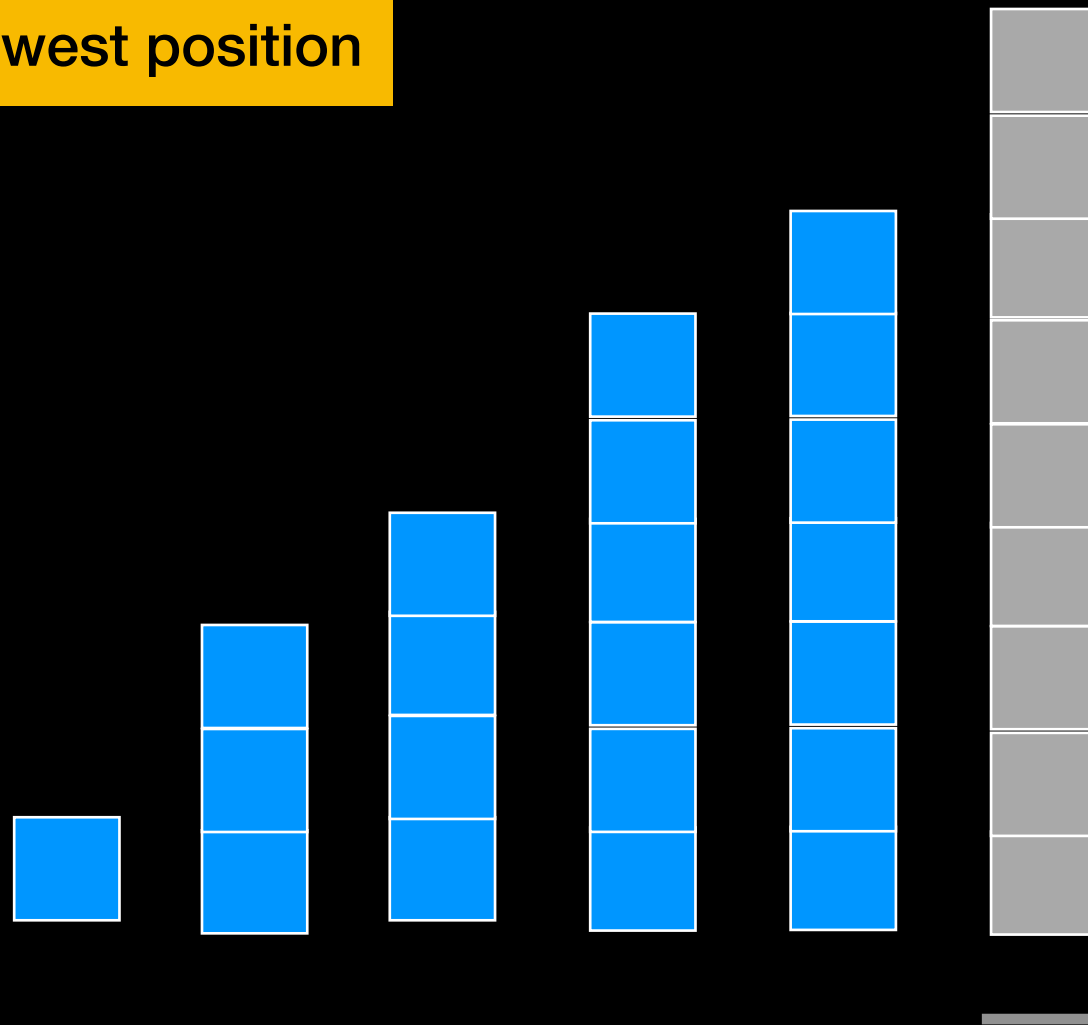
Selection Sort




Find smallest element and
move it at lowest position

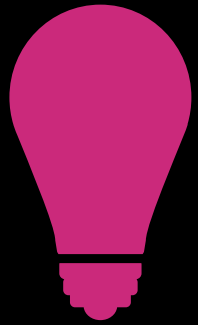


6th Pass

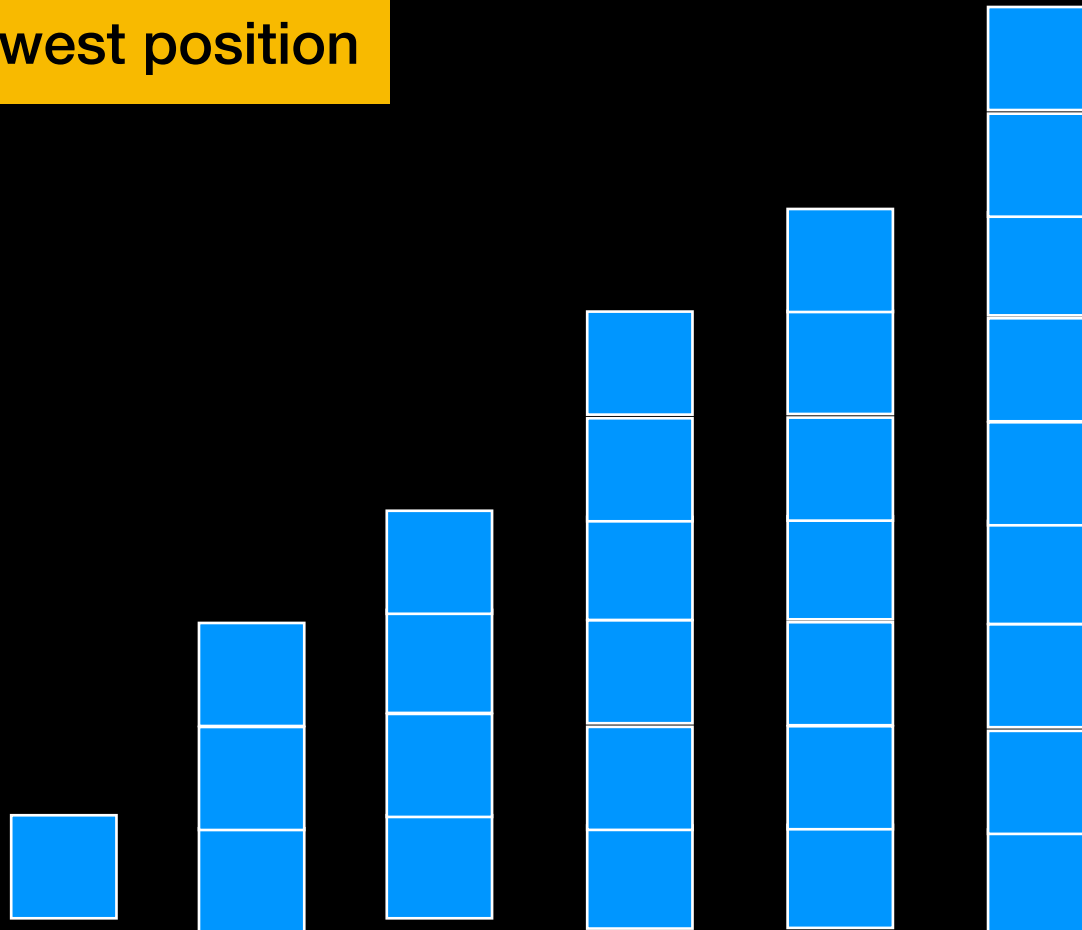


Selection Sort

 Unsorted
 Sorted



Find smallest element and
move it at lowest position



Selection Sort

Find the smallest item and move it at position 1

Find the next-smallest item and move it at position 2

...

Selection Sort Analysis

How much work?

Find smallest: look at **n** elements

Selection Sort Analysis

How much work?

Find smallest: look at **n** elements

Find second smallest: look at **$n-1$** elements

Selection Sort Analysis

How much work?

Find smallest: look at n elements

Find second smallest: look at $n-1$ elements

Find third smallest: look at $n-2$ elements

...

Selection Sort Analysis

How much work?

Find smallest: look at n elements

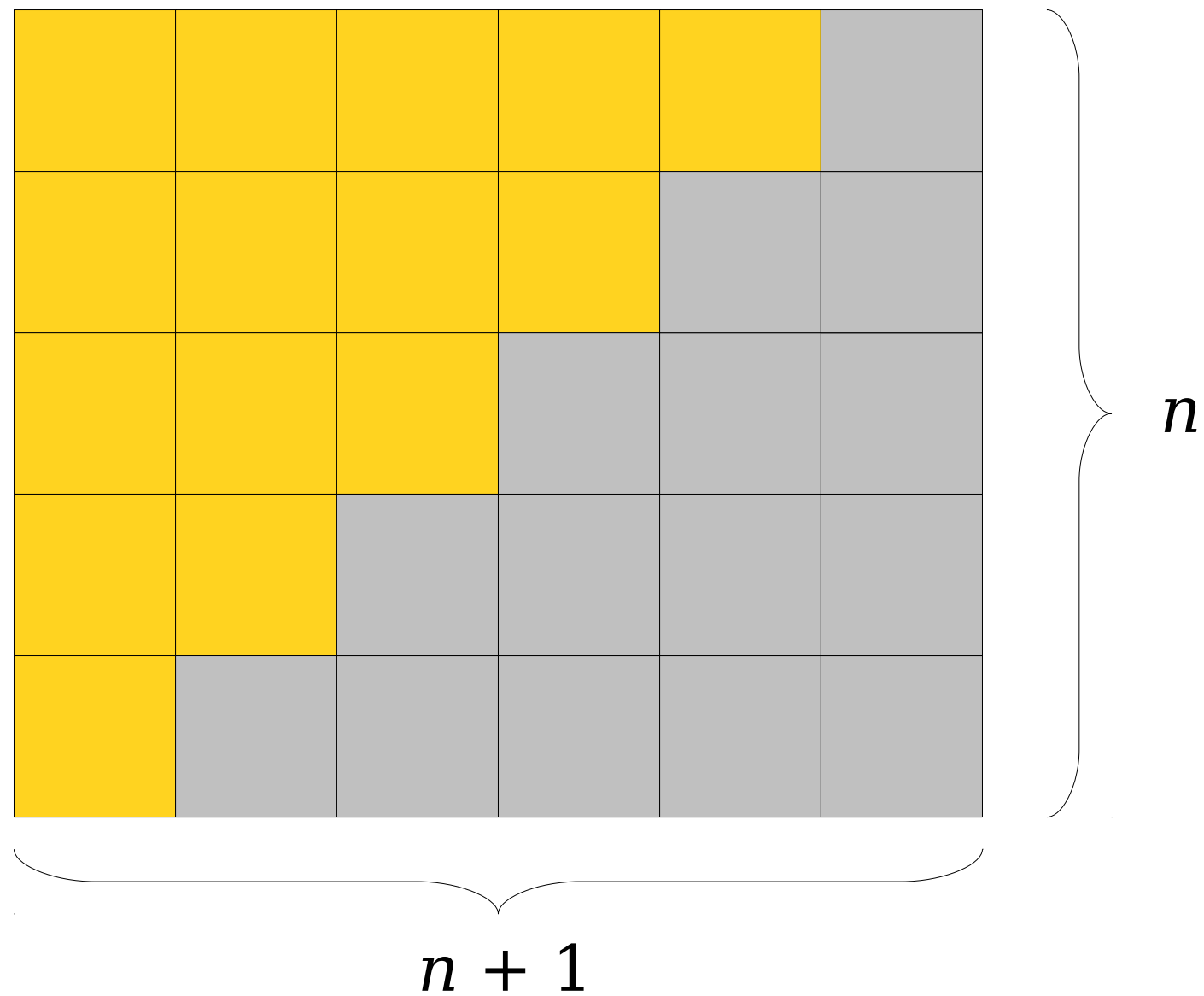
Find second smallest: look at $n-1$ elements

Find third smallest: look at $n-2$ elements

...

Total work: $n + (n-1) + (n-2) + \dots + 1$

$$n + (n-1) + \dots + 2 + 1 = n(n+1) / 2$$



Selection Sort Analysis

$$T(n) = n(n+1) / 2 \text{ comparisons} + n \text{ data moves} = O(\text{ })?$$

Selection Sort Analysis

$$T(n) = n(n+1) / 2 \text{ comparisons} + n \text{ data moves} = O(\text{ })?$$

$$T(n) = (n^2+n) / 2 + n = O(\text{ })?$$

Selection Sort Analysis

$$T(n) = n(n+1) / 2 \text{ comparisons} + n \text{ data moves} = O(\text{ })?$$

$$T(n) = (n^2+n) / 2 + n = O(\text{ })?$$

Ignore constant

Ignore non-dominant terms

Selection Sort Analysis

$$T(n) = n(n+1) / 2 \text{ comparisons} + n \text{ data moves} = O(\text{ })?$$

$$T(n) = (n^2+n) / 2 + n = O(n^2)$$

Ignore constant

Ignore non-dominant terms

Selection Sort Analysis

$$T(n) = n(n+1) / 2 \text{ comparisons} + n \text{ data moves} = O(\text{ })?$$

$$T(n) = (n^2+n) / 2 + n = O(n^2)$$

Selection Sort run time is $O(n^2)$

```

template<typename ItemType>
void selectionSort(ItemType the_array[], size_t size)
{
    // first = index of the first item in the subarray of items yet
    //           to be sorted;
    // smallest = index of the smallest item found
    for (int first = 0; first < size; first++)
    {
        // At this point, the_array[0 ...first-1] is sorted, and its
        // entries are <= those in the_array[first ... size-1].
        // Select the smallest entry in the_array[first ... size-1]
        int smallest_index = findIndexOfSmallest(the_array, first,
                                                size);

        // Swap the smallest entry, the_array[smallest_index], with
        // the first in the unsorted subarray the_array[first]
        swap(the_array[smallest_index], the_array[first]);
    } // end for
} // end selectionSort

```

```

template<typename ItemType>
void selectionSort(ItemType the_array[], size_t size)
{
    // first = index of the first item in the subarray of items yet
    //           to be sorted;
    // smallest = index of the smallest item found
Pass for (int first = 0; first < size; first++)
O(n) {
    // At this point, the_array[0 ...first-1] is sorted, and its
    // entries are <= those in the_array[first ... size-1].
    // Select the smallest entry in the_array[first ... size-1]
O(n) int smallest_index = findIndexOfSmallest(the_array, first,
                                                size);
    // Swap the smallest entry, the_array[smallest_index], with
    // the first in the unsorted subarray the_array[first]
    swap(the_array[smallest_index], the_array[first]);
    } // end for
} // end selectionSort


```

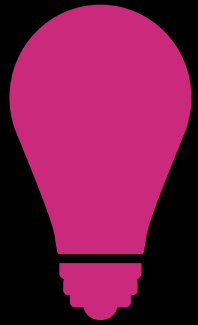
$O(n^2)$

Stability

A sorting algorithm is **Stable** if elements that are equal remain in same order relative to each other after sorting

Selection Sort


 Unsorted
 Sorted

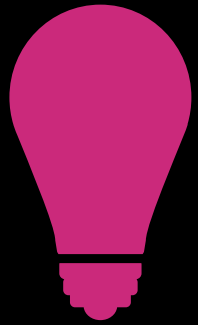


Find smallest element and
move it at lowest position



Selection Sort

 Unsorted
 Sorted

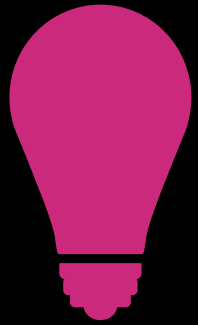


Find smallest element and
move it at lowest position

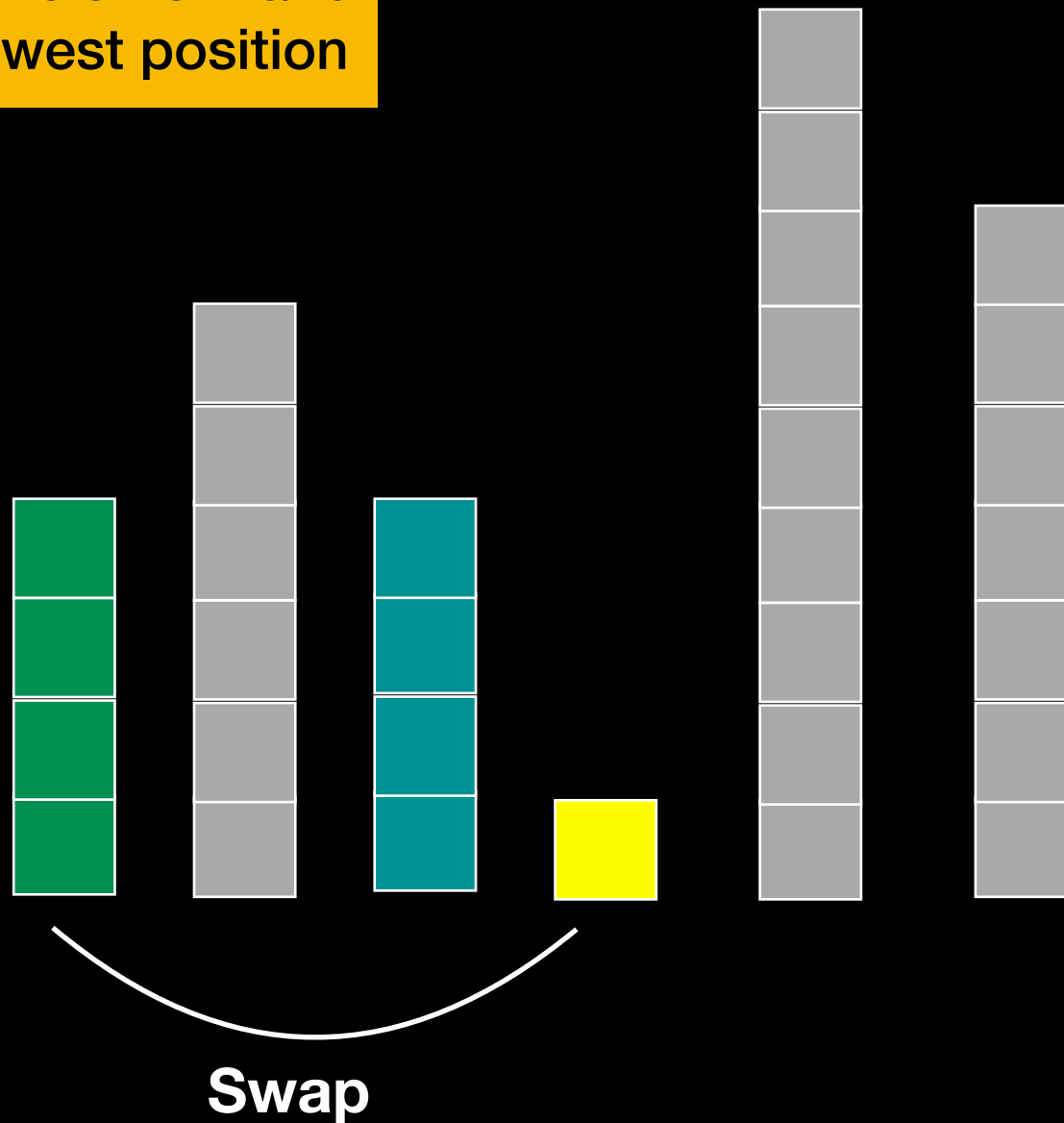


Selection Sort

Unsorted
Sorted

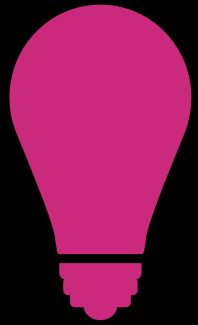


Find smallest element and
move it at lowest position

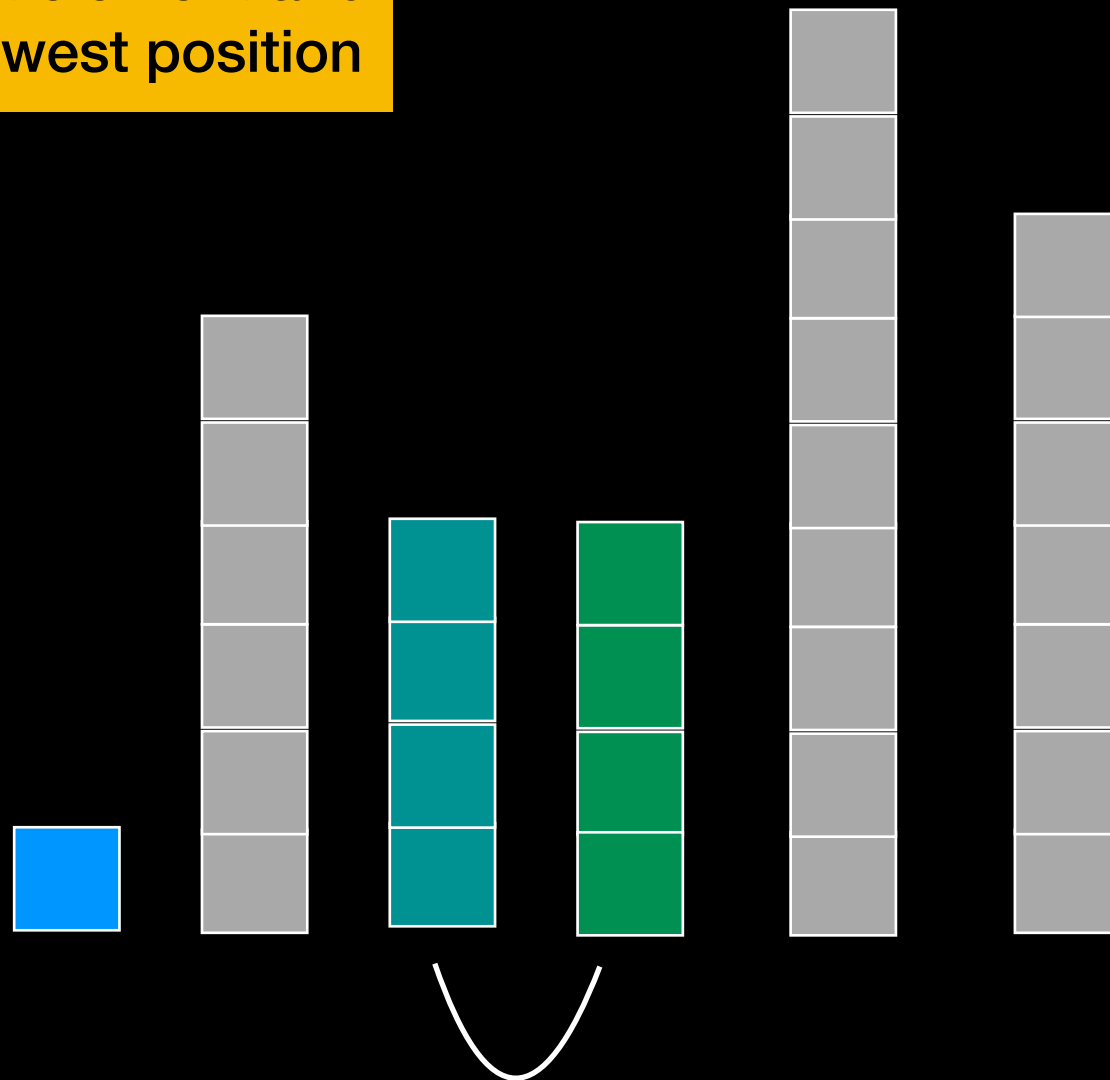


Selection Sort

Unsorted
Sorted



Find smallest element and
move it at lowest position



Unstable

Selection Sort Analysis

Execution time DOES NOT depend on initial arrangement of data => ALWAYS $O(n^2)$

$O(n^2)$ comparisons

Good choice for small n and/or data moves are costly
($O(n)$ data moves)

Unstable

Understanding $O(n^2)$

100	14	3	43	200	274
-----	----	---	----	-----	-----

$T(n)$

Understanding $O(n^2)$

100	14	3	43	200	274
-----	----	---	----	-----	-----

$T(n)$

100	14	3	43	200	274	523	108	76	195	599	158
-----	----	---	----	-----	-----	-----	-----	----	-----	-----	-----

$$T(2n) \approx 4T(n)$$

$$(2n)^2 = 4n^2$$

Understanding $O(n^2)$

100	14	3	43	200	274
-----	----	---	----	-----	-----

$T(n)$

100	14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932	5
-----	----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----	---

$$T(3n) \approx 9T(n)$$

$$(3n)^2 = 9n^2$$

Understanding $O(n^2)$ on large input

If size of **input** increases by factor of **100**

Execution time increases by factor of **10,000**

$$T(100n) = 10,000T(n)$$

Understanding $O(n^2)$ on large input

If size of **input** increases by factor of **100**

Execution time increases by factor of **10,000**

$$T(100n) = 10,000T(n)$$

Assume $n = 100,000$ and $T(n) = 17$ seconds

Sorting **10,000,000** takes **10,000** longer

Understanding $O(n^2)$ on large input

If size of **input** increases by factor of **100**

Execution time increases by factor of **10,000**

$$T(100n) = 10,000T(n)$$

Assume $n = 100,000$ and $T(n) = 17$ seconds

Sorting **10,000,000** takes **10,000** longer

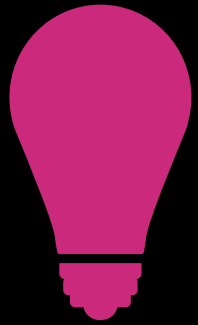
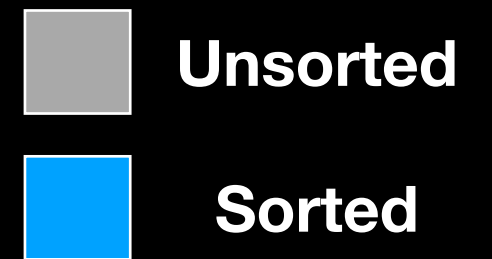
Sorting **10,000,000** entries takes \approx **2 days**

Multiplying input by **100** to go from **17sec** to **2 days!!!**

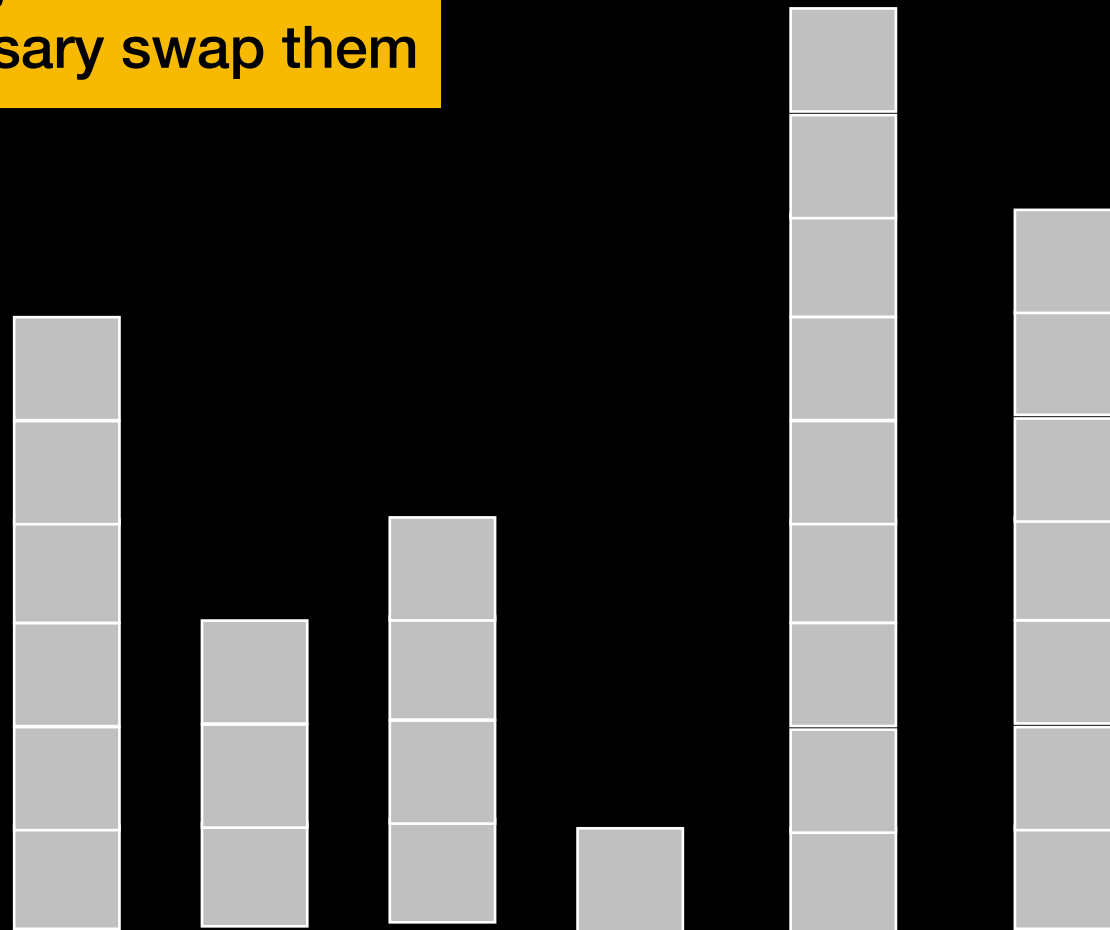
Raise your hand if you had
Selection Sort

Bubble Sort

Bubble Sort



Compare adjacent elements
and if necessary swap them



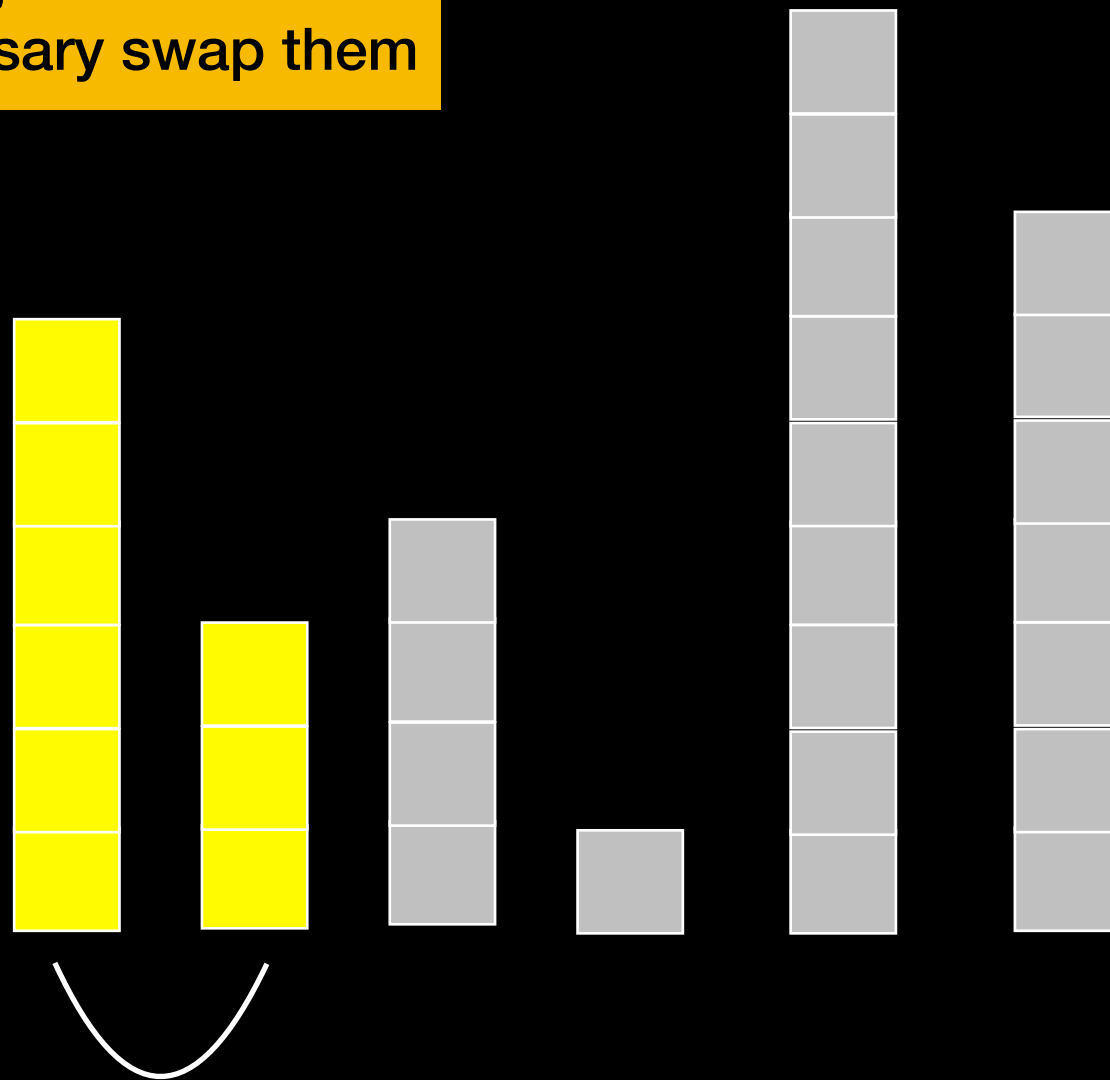
Bubble Sort



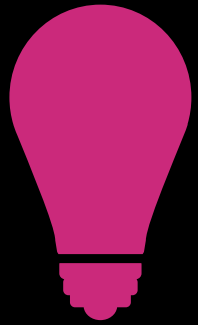
Compare adjacent elements
and if necessary swap them



1st Pass



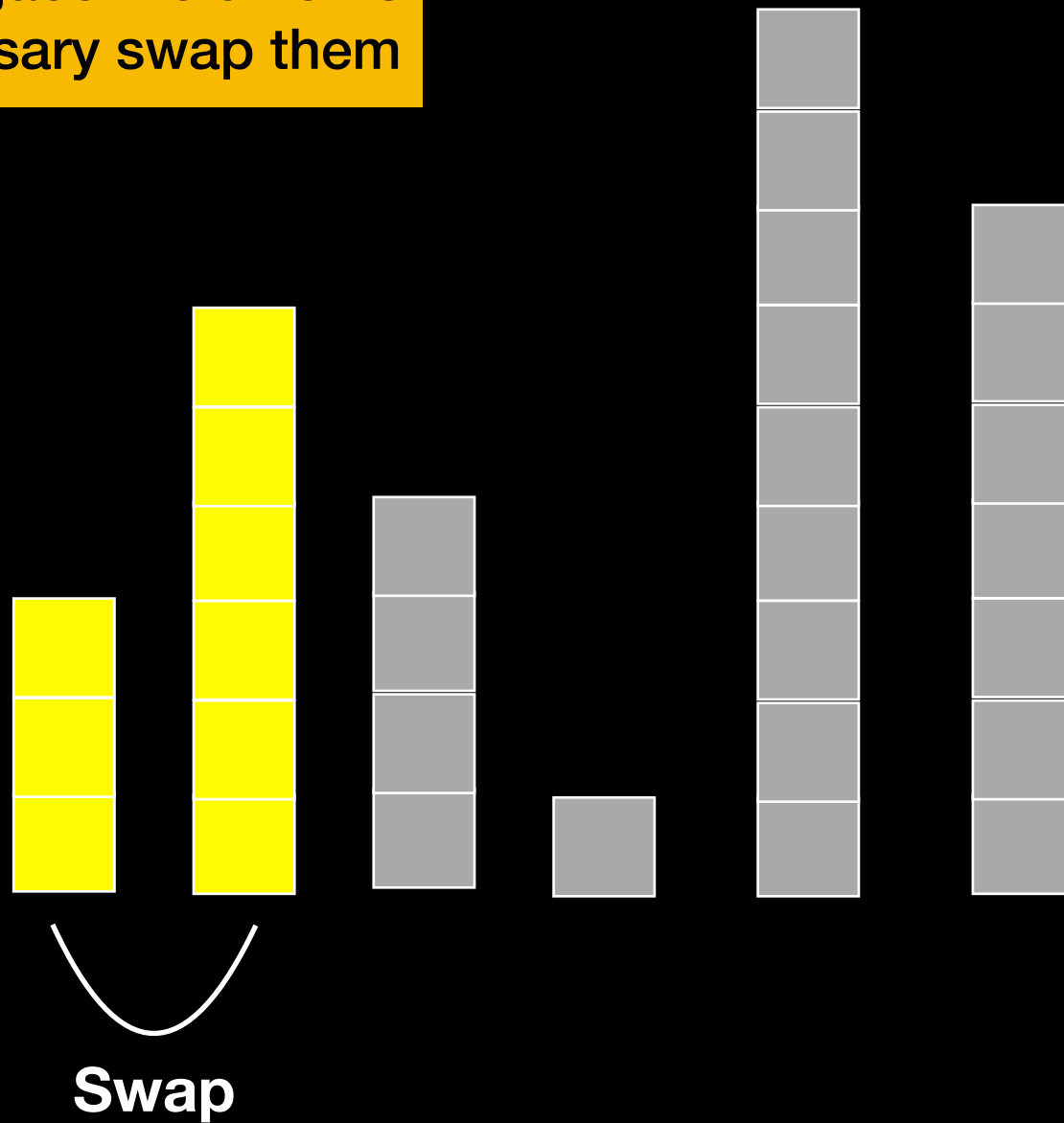
Bubble Sort



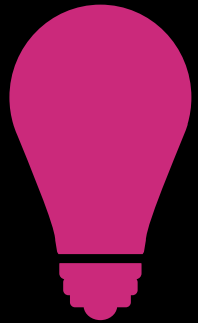
Compare adjacent elements
and if necessary swap them



1st Pass



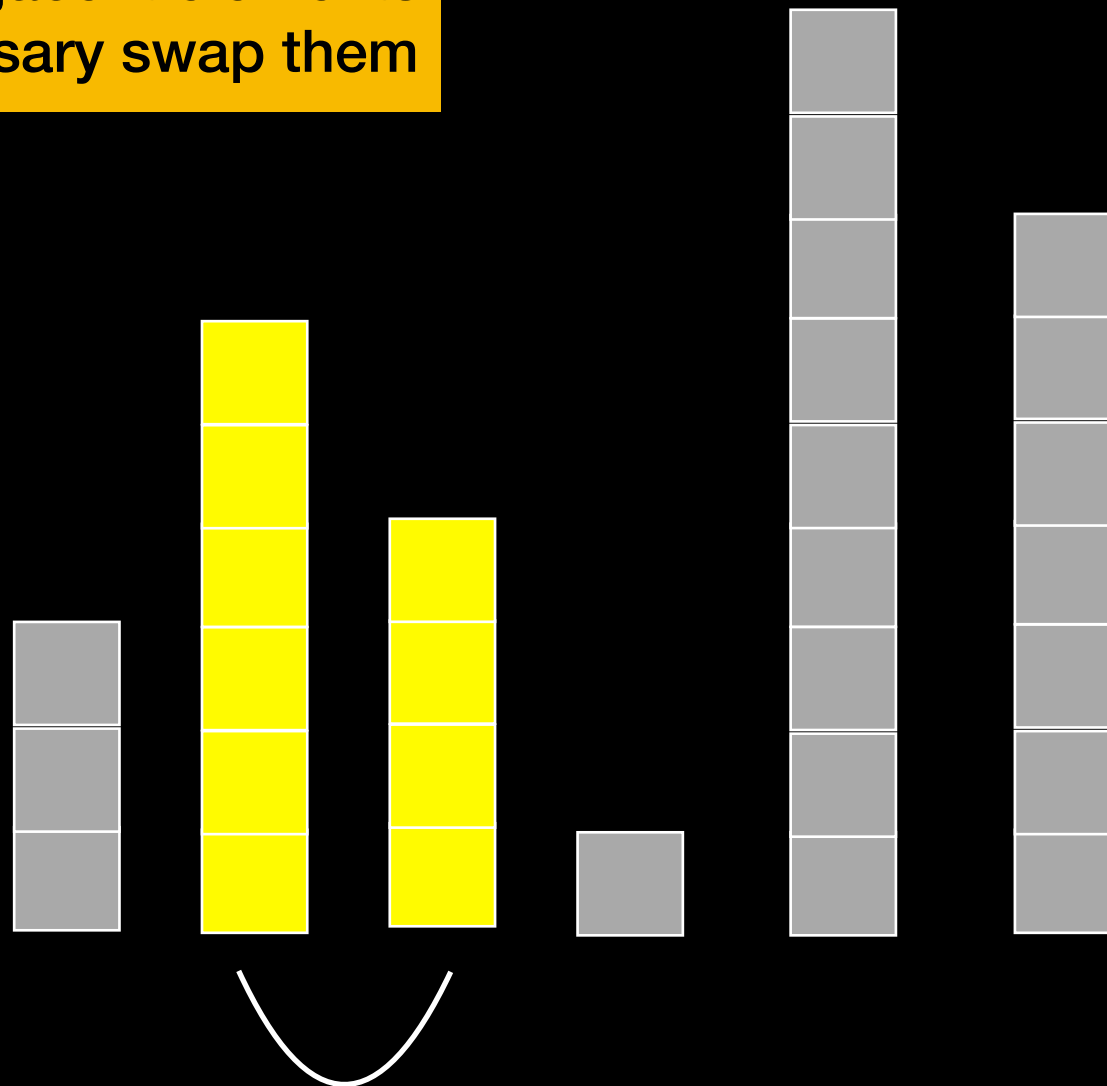
Bubble Sort



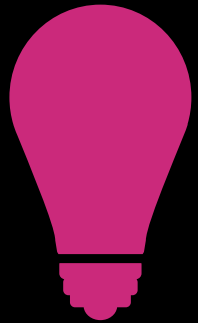
Compare adjacent elements
and if necessary swap them



1st Pass



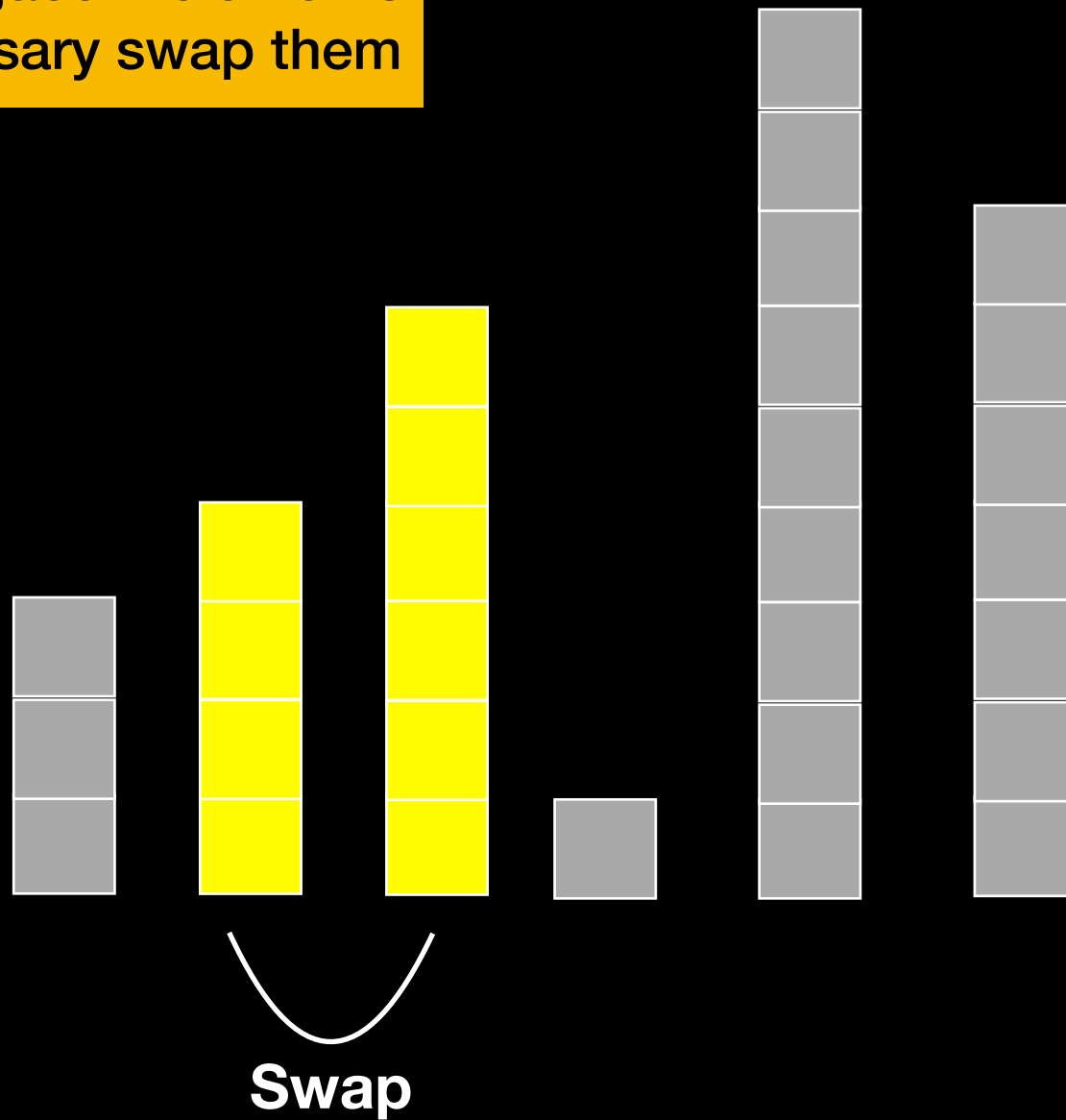
Bubble Sort



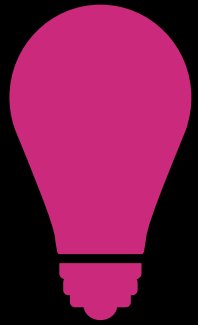
Compare adjacent elements
and if necessary swap them



1st Pass



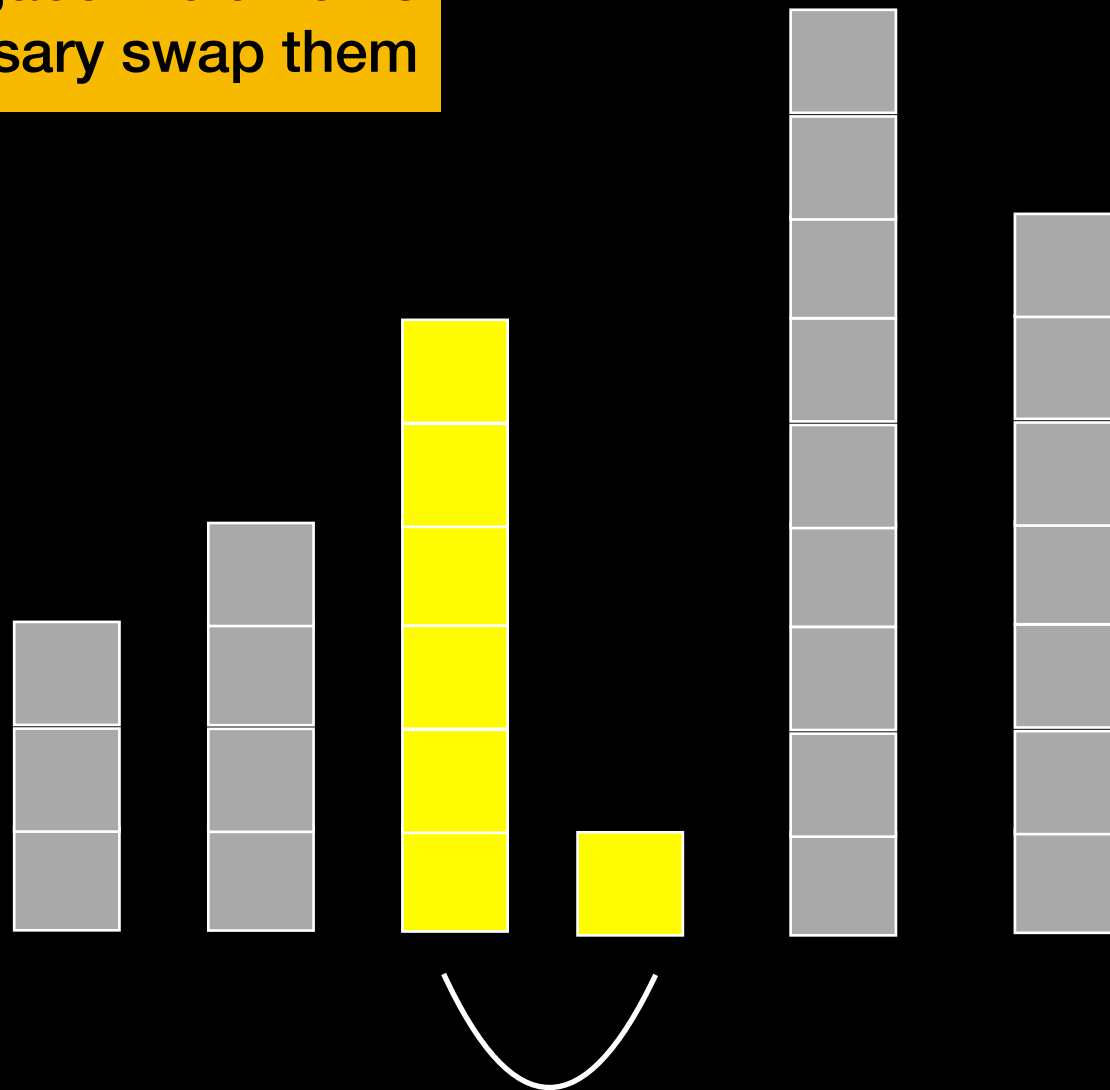
Bubble Sort



Compare adjacent elements
and if necessary swap them



1st Pass



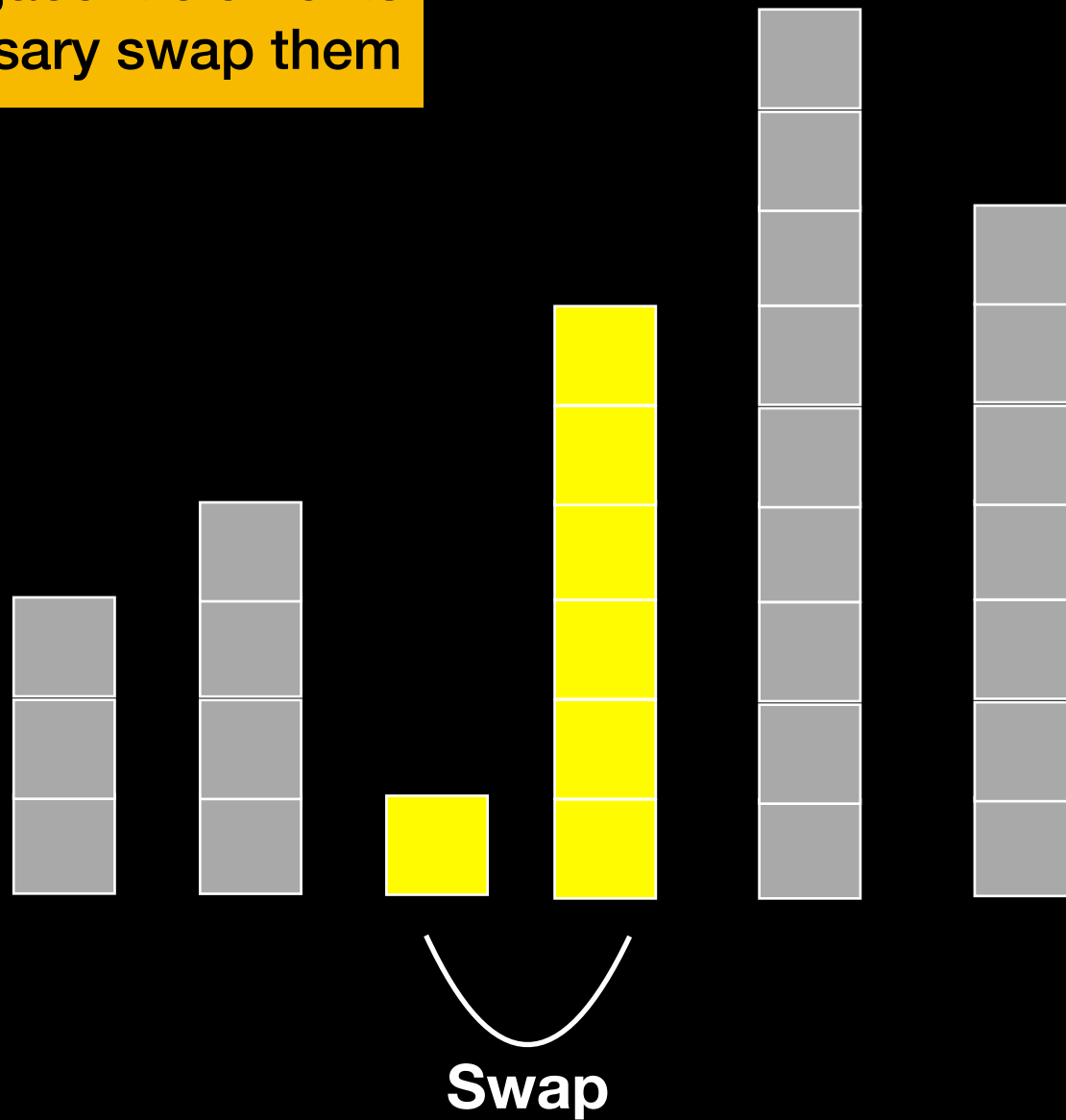
Bubble Sort



Compare adjacent elements
and if necessary swap them



1st Pass



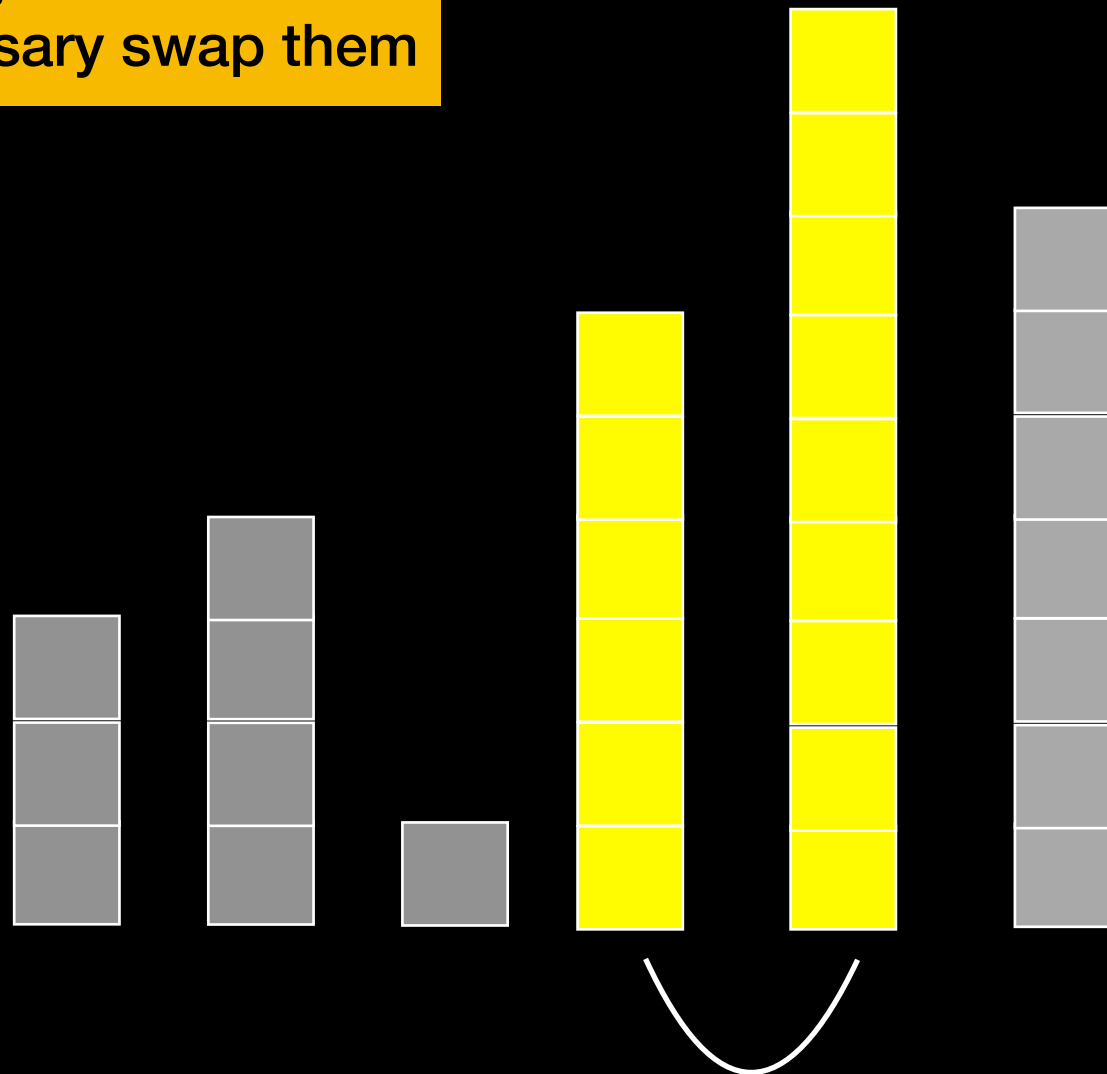
Bubble Sort



Compare adjacent elements
and if necessary swap them



1st Pass



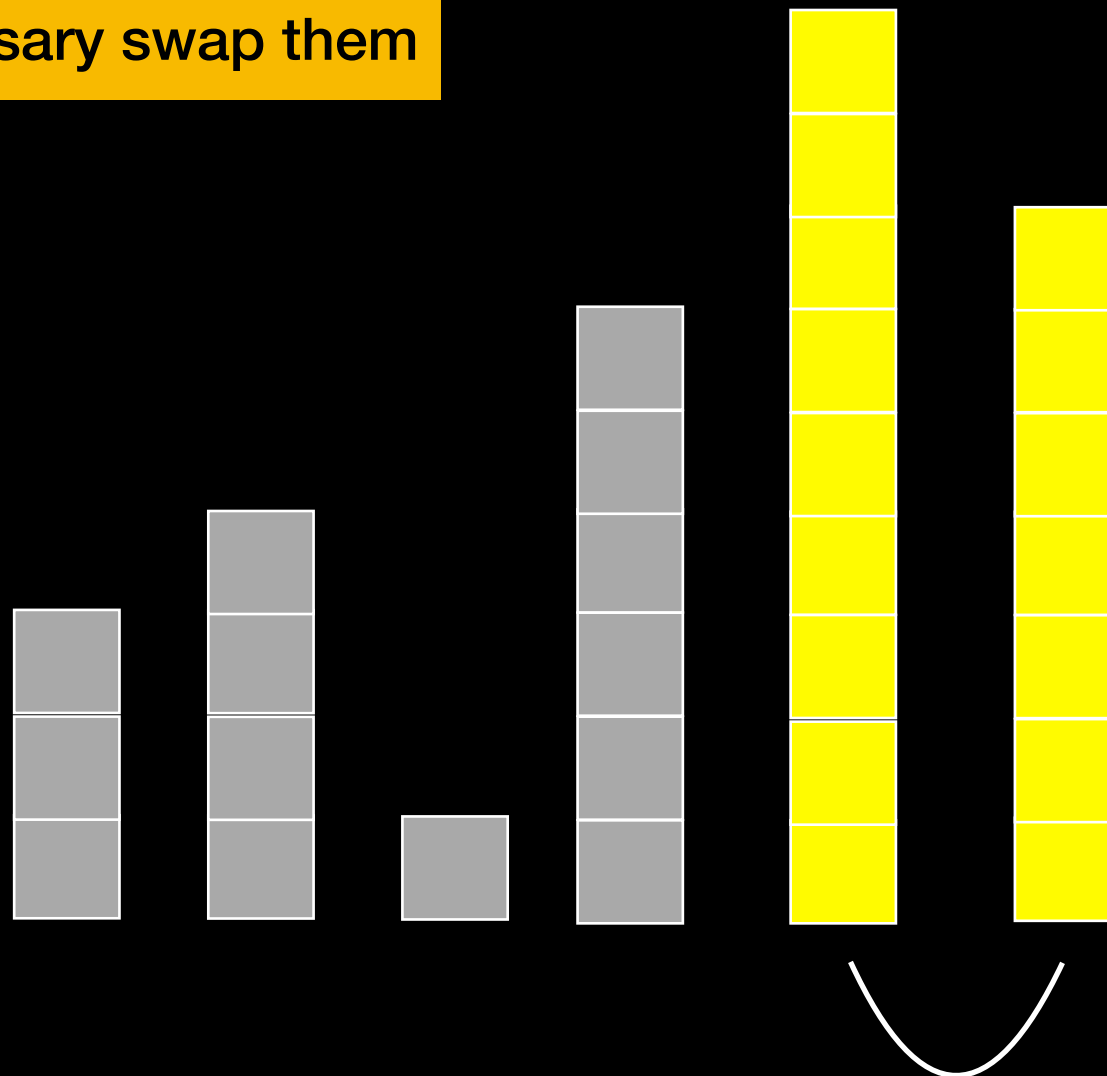
Bubble Sort



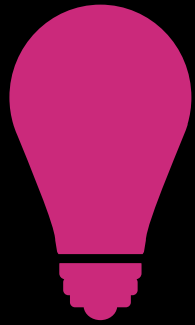
Compare adjacent elements
and if necessary swap them



1st Pass



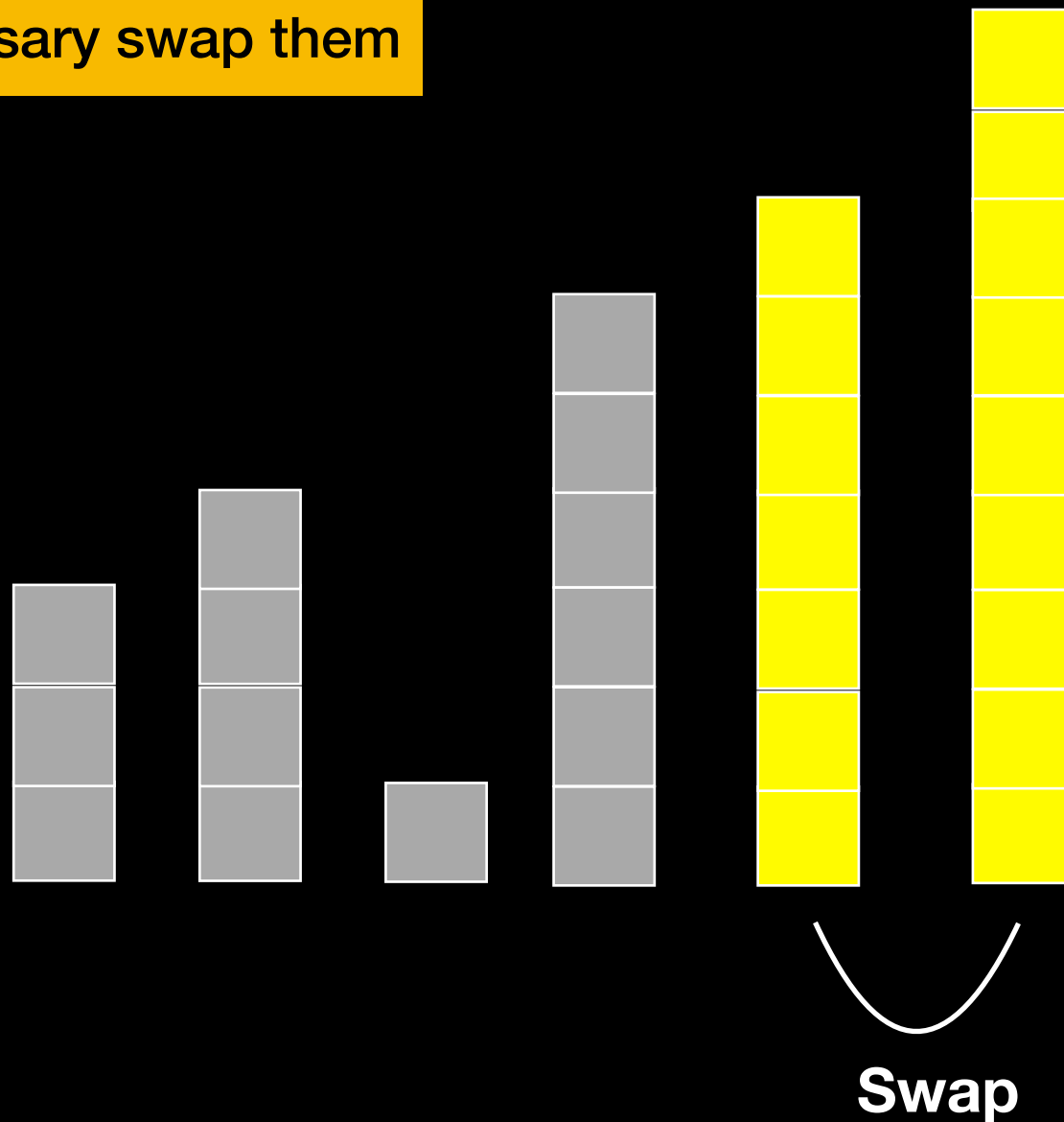
Bubble Sort



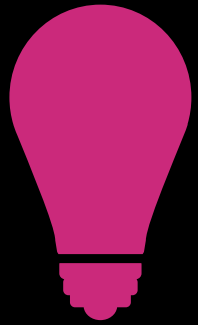
Compare adjacent elements
and if necessary swap them



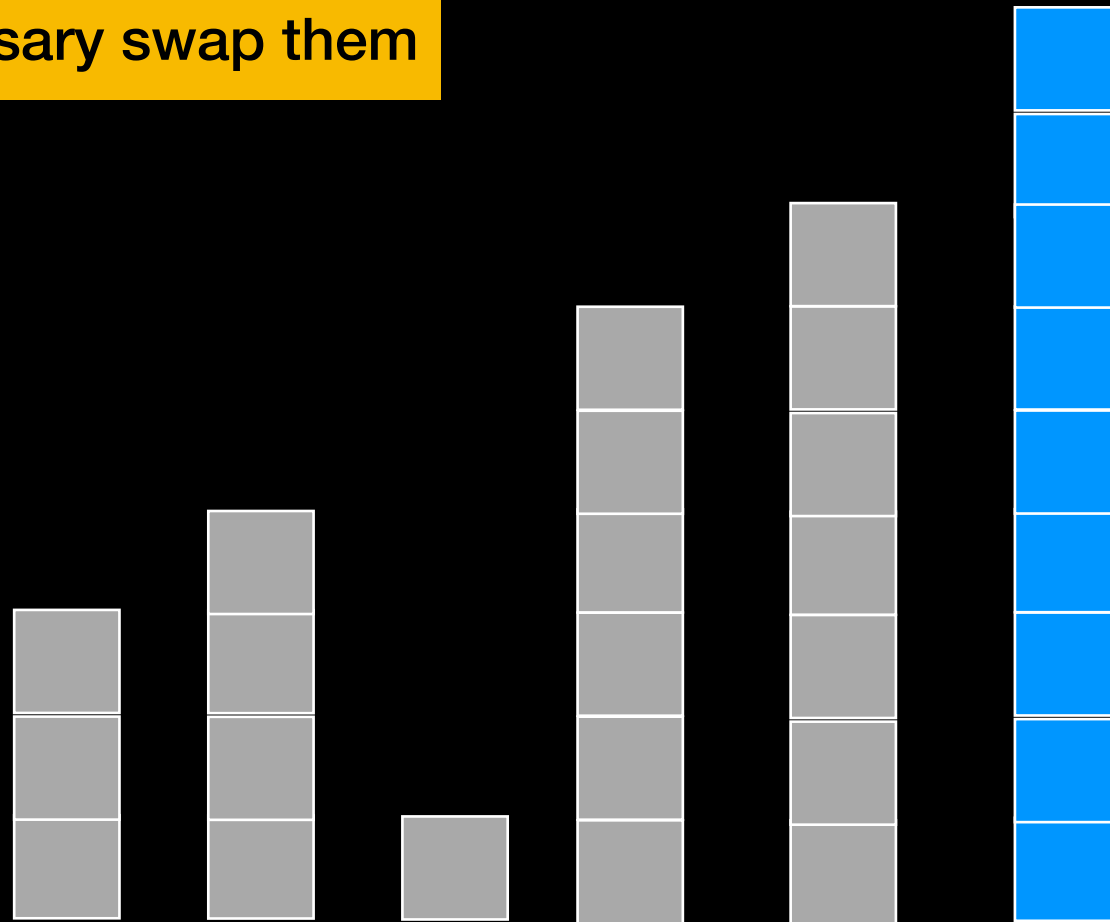
1st Pass



Bubble Sort



Compare adjacent elements
and if necessary swap them



End of 1st Pass:

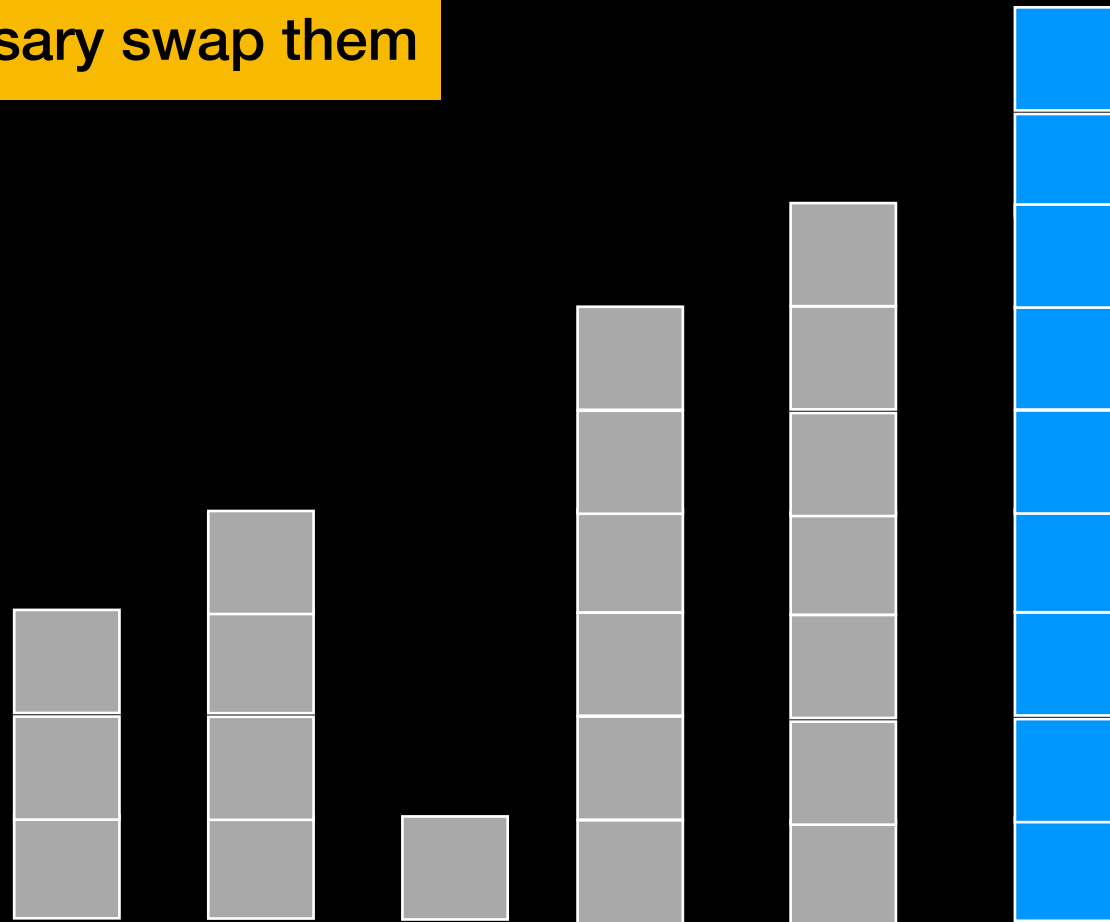
Not sorted, but largest has
"bubbled up" to its proper
position

Bubble Sort

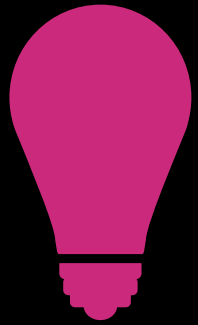


Compare adjacent elements
and if necessary swap them

2nd Pass:
Sort **n-1**



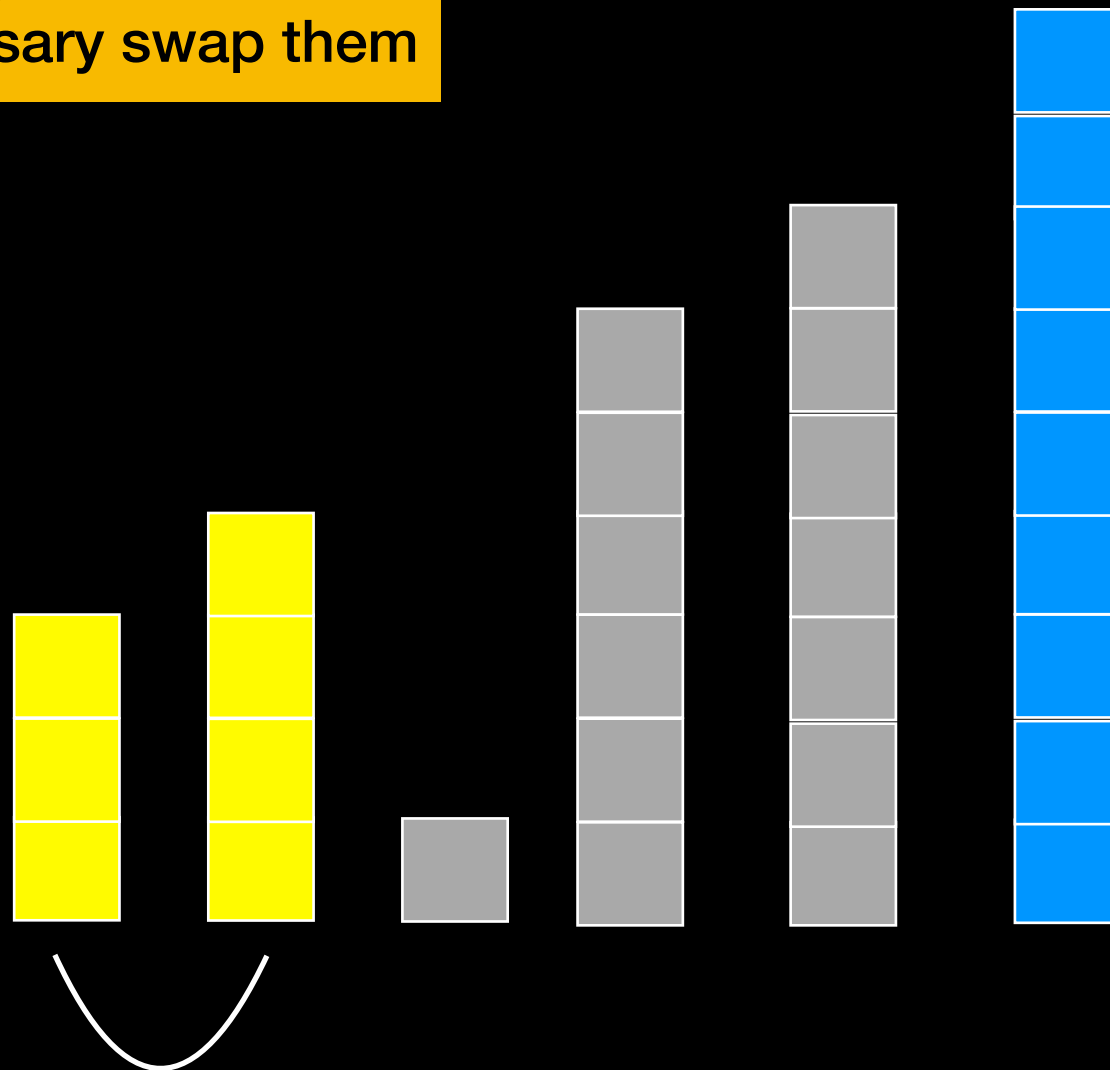
Bubble Sort



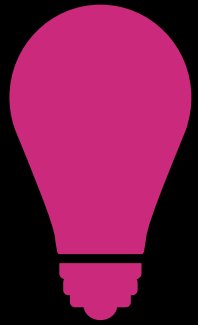
Compare adjacent elements
and if necessary swap them



2nd Pass



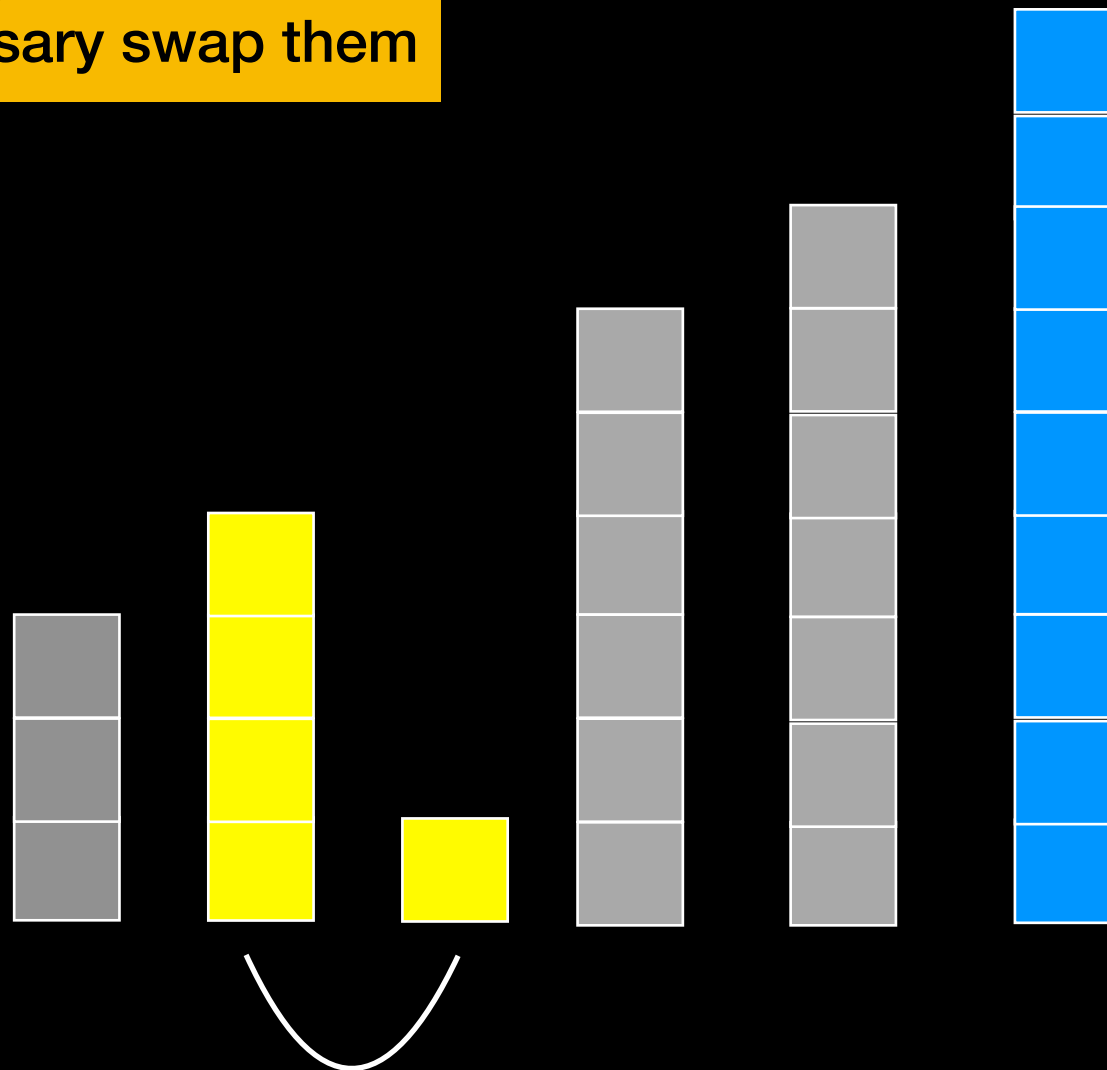
Bubble Sort



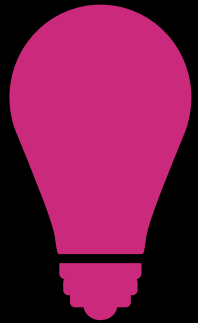
Compare adjacent elements
and if necessary swap them



2nd Pass



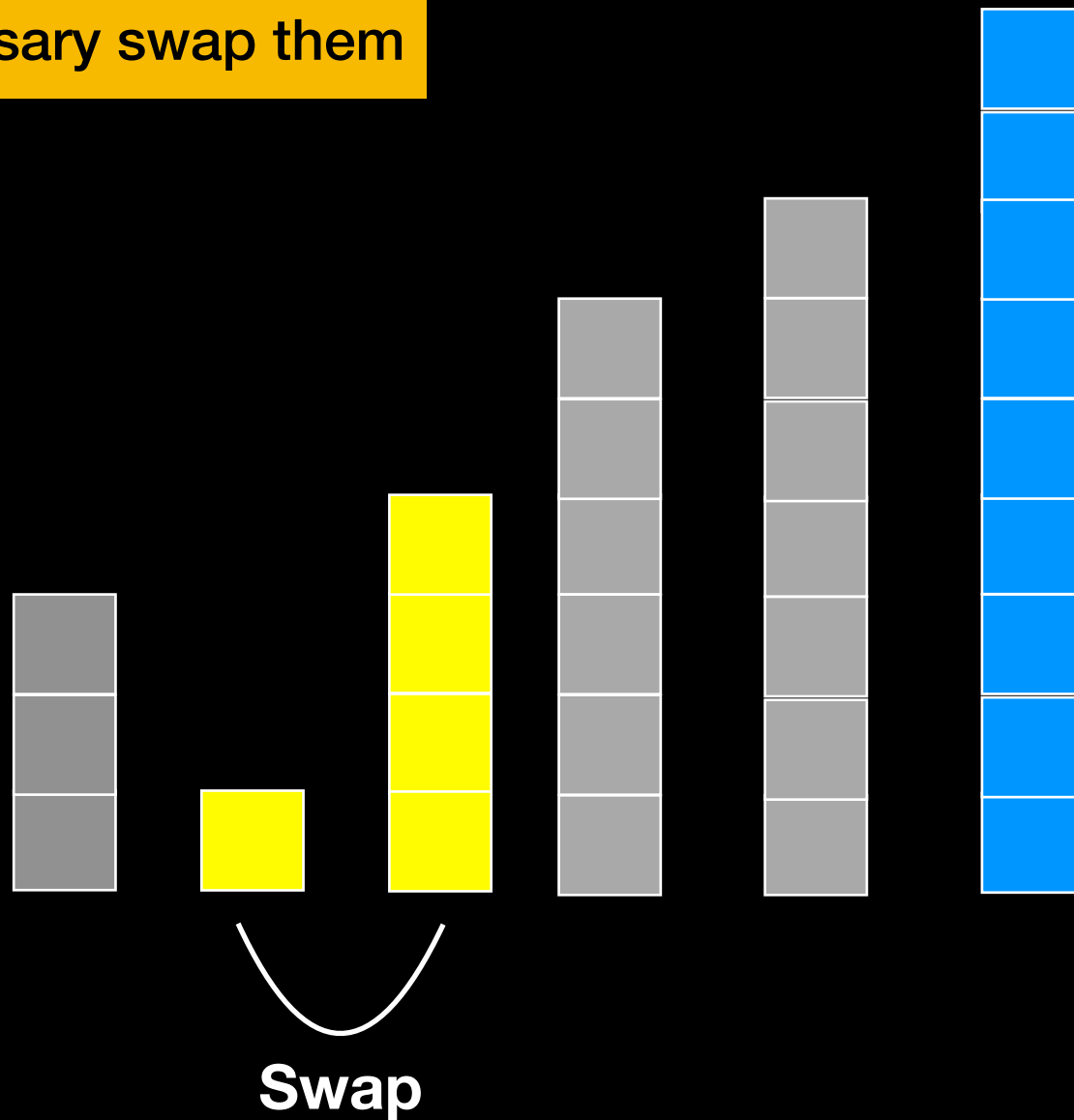
Bubble Sort



Compare adjacent elements
and if necessary swap them



2nd Pass



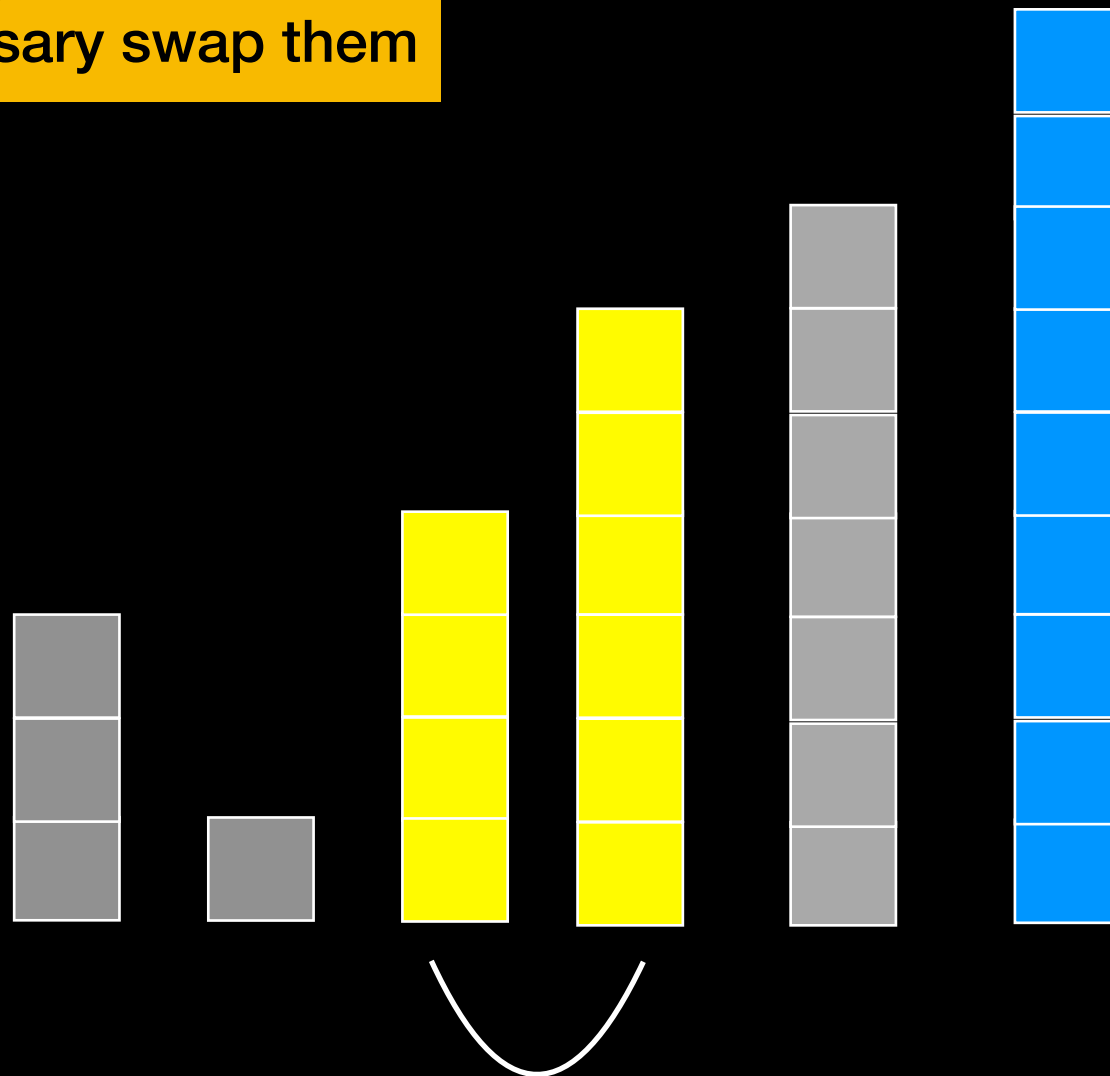
Bubble Sort



Compare adjacent elements
and if necessary swap them



2nd Pass



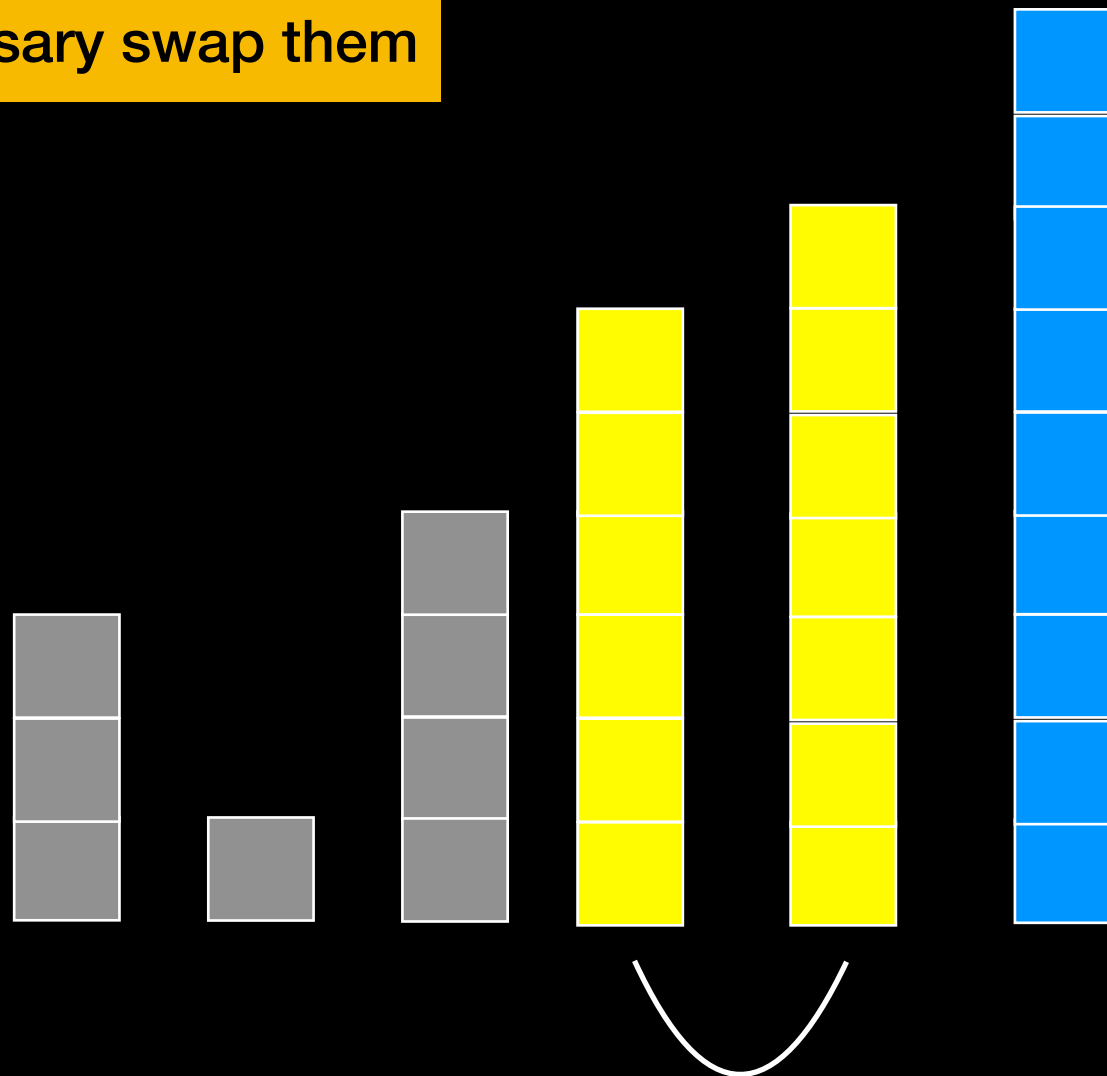
Bubble Sort



Compare adjacent elements
and if necessary swap them



2nd Pass

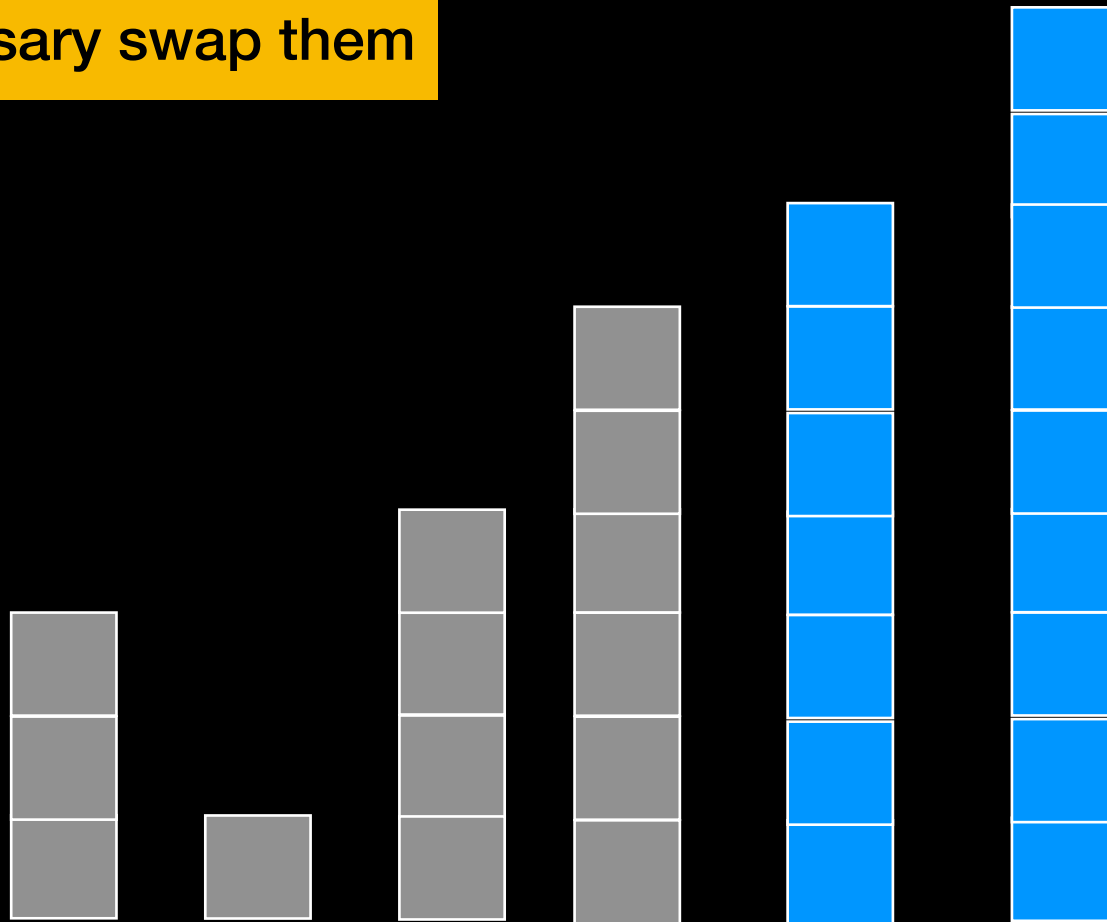


Bubble Sort

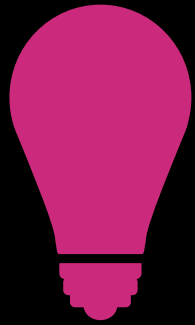


Compare adjacent elements
and if necessary swap them

3rd Pass:
Sort **n-2**



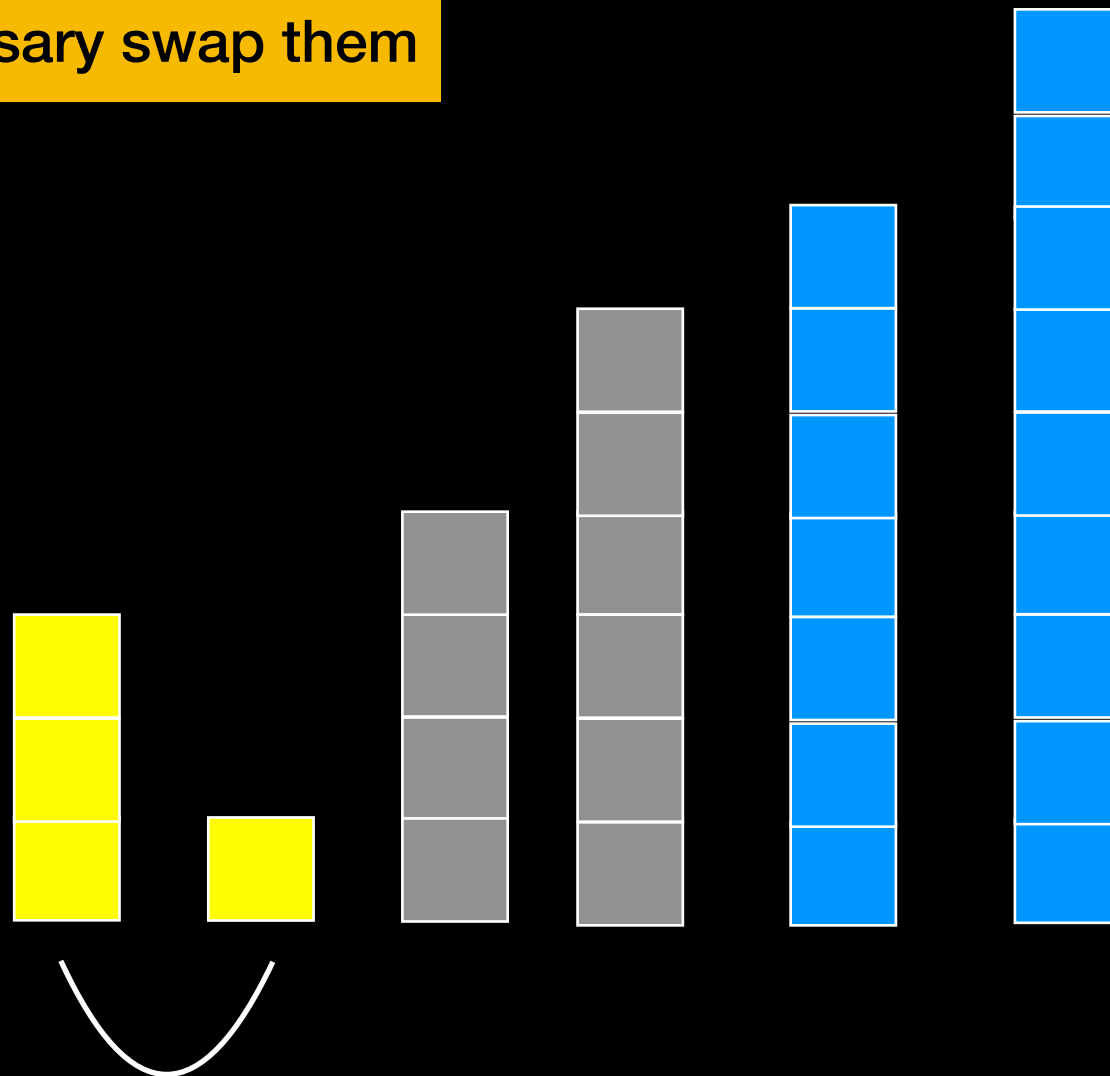
Bubble Sort



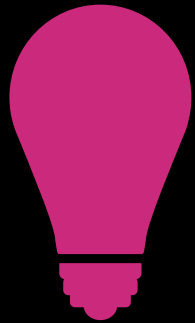
Compare adjacent elements
and if necessary swap them



3rd Pass



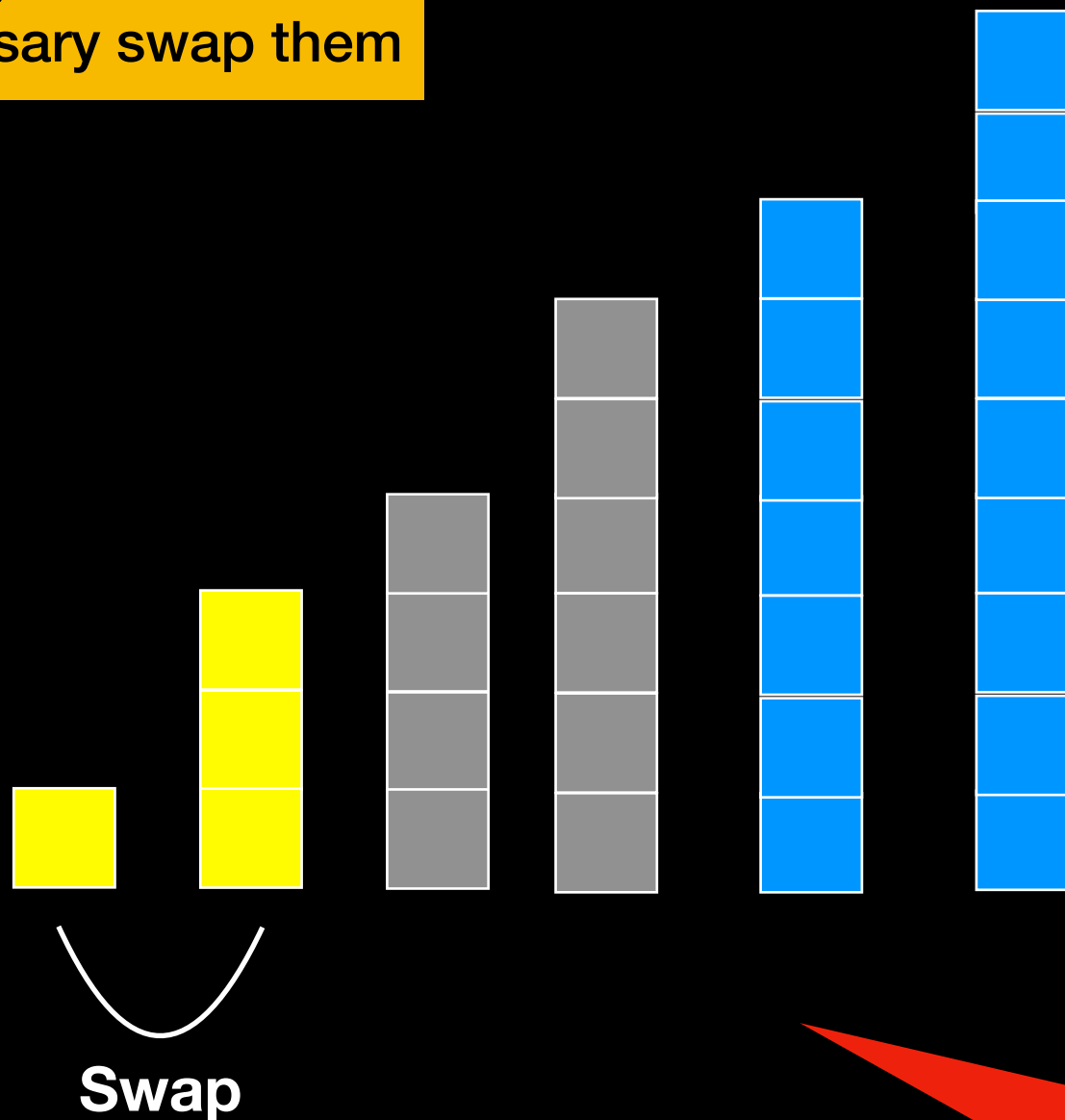
Bubble Sort



Compare adjacent elements
and if necessary swap them

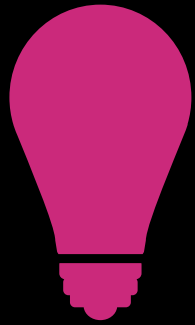
Unsorted
Sorted

3rd Pass



Array is sorted
But our algorithm doesn't know
It keeps on going

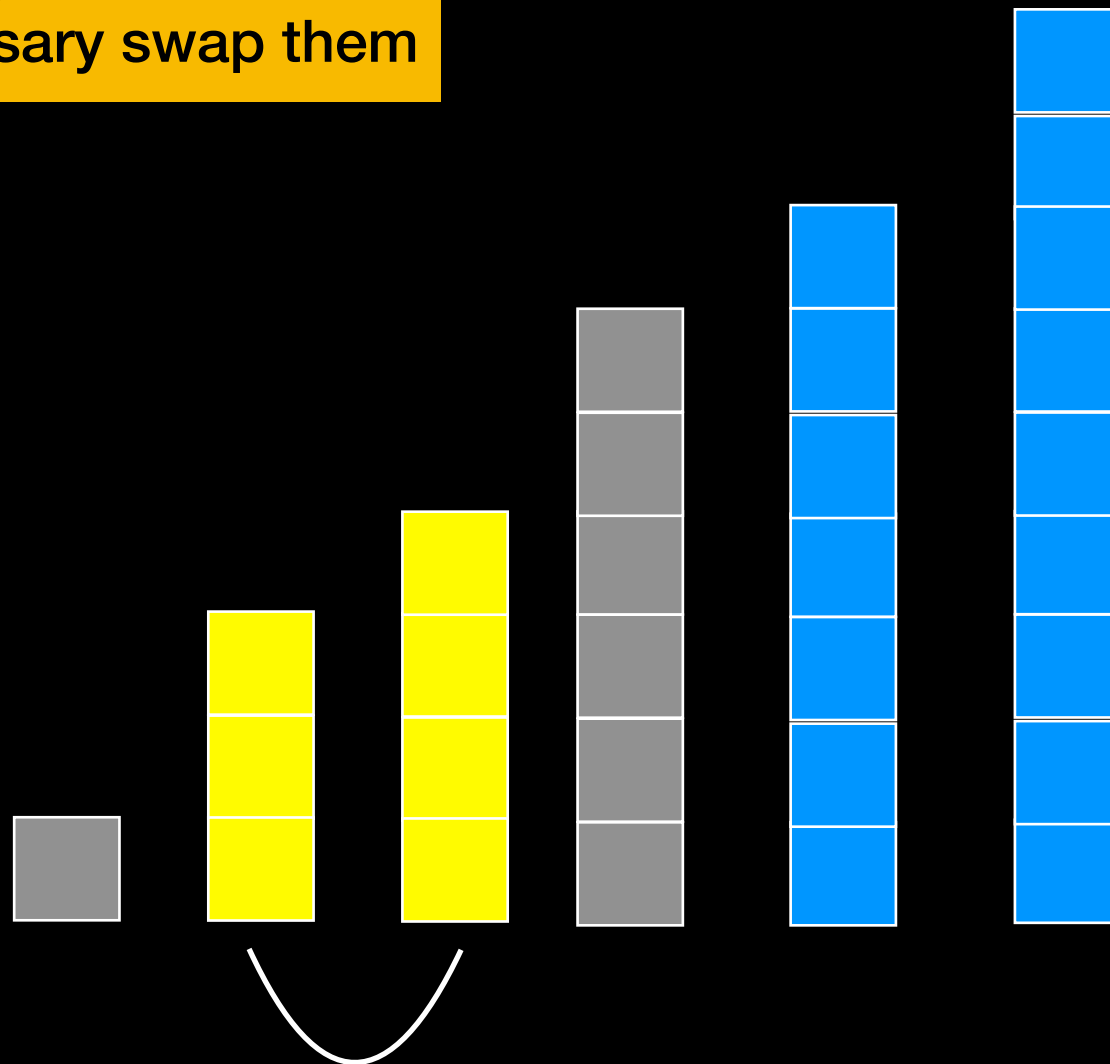
Bubble Sort



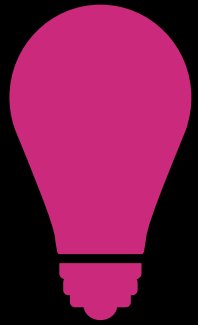
Compare adjacent elements
and if necessary swap them



3rd Pass



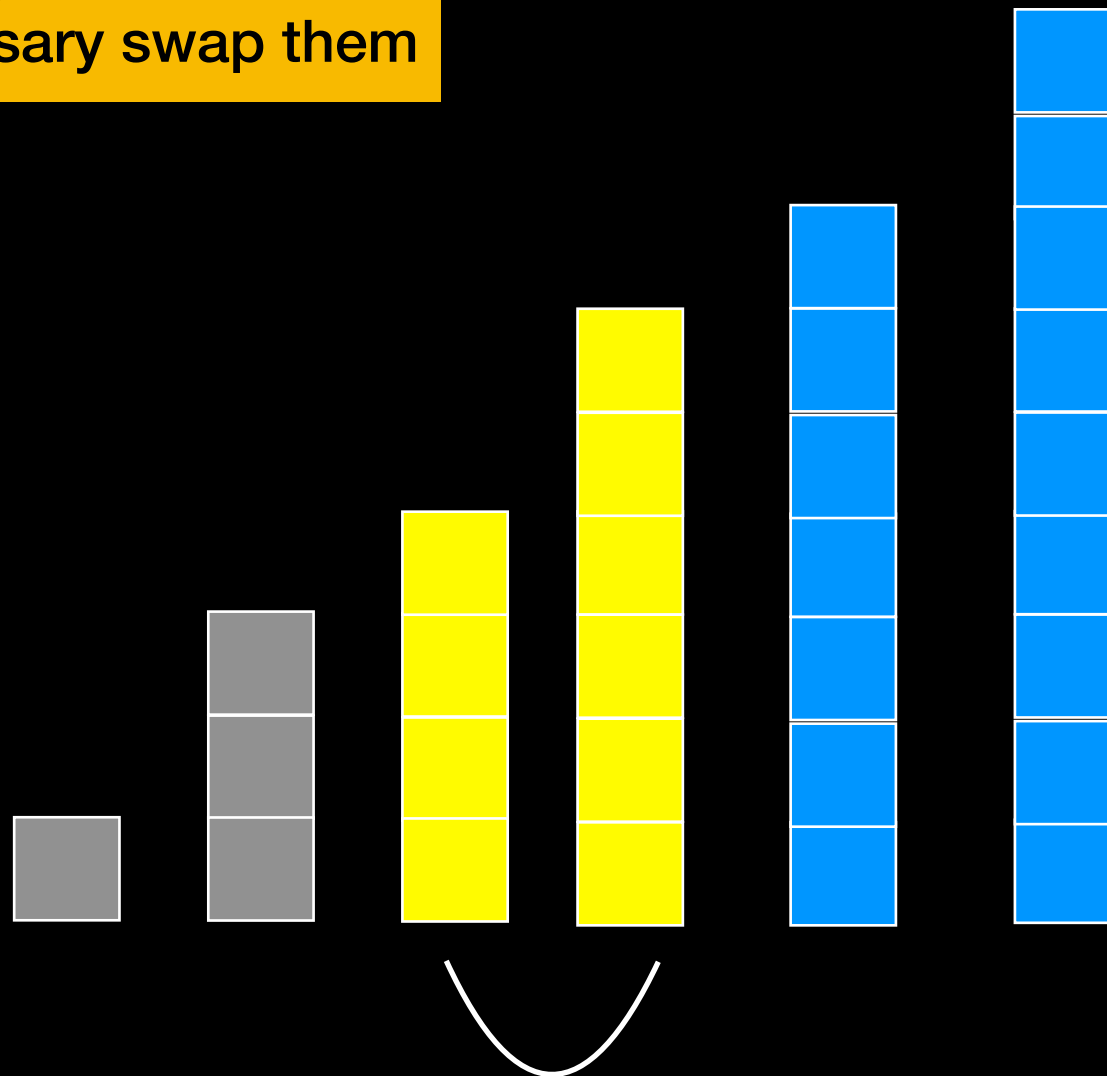
Bubble Sort



Compare adjacent elements
and if necessary swap them



3rd Pass

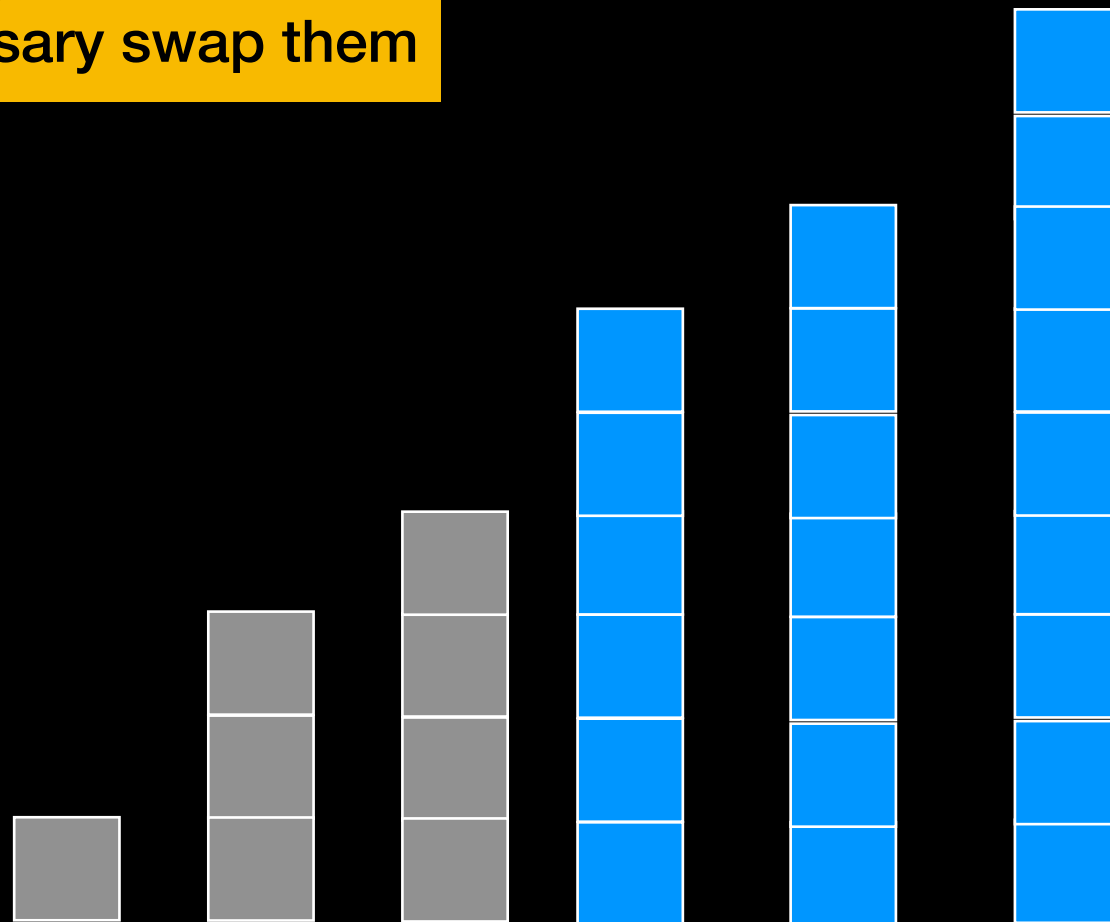


Bubble Sort

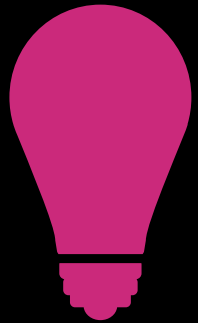


Compare adjacent elements
and if necessary swap them

4th Pass:
Sort **n-3**



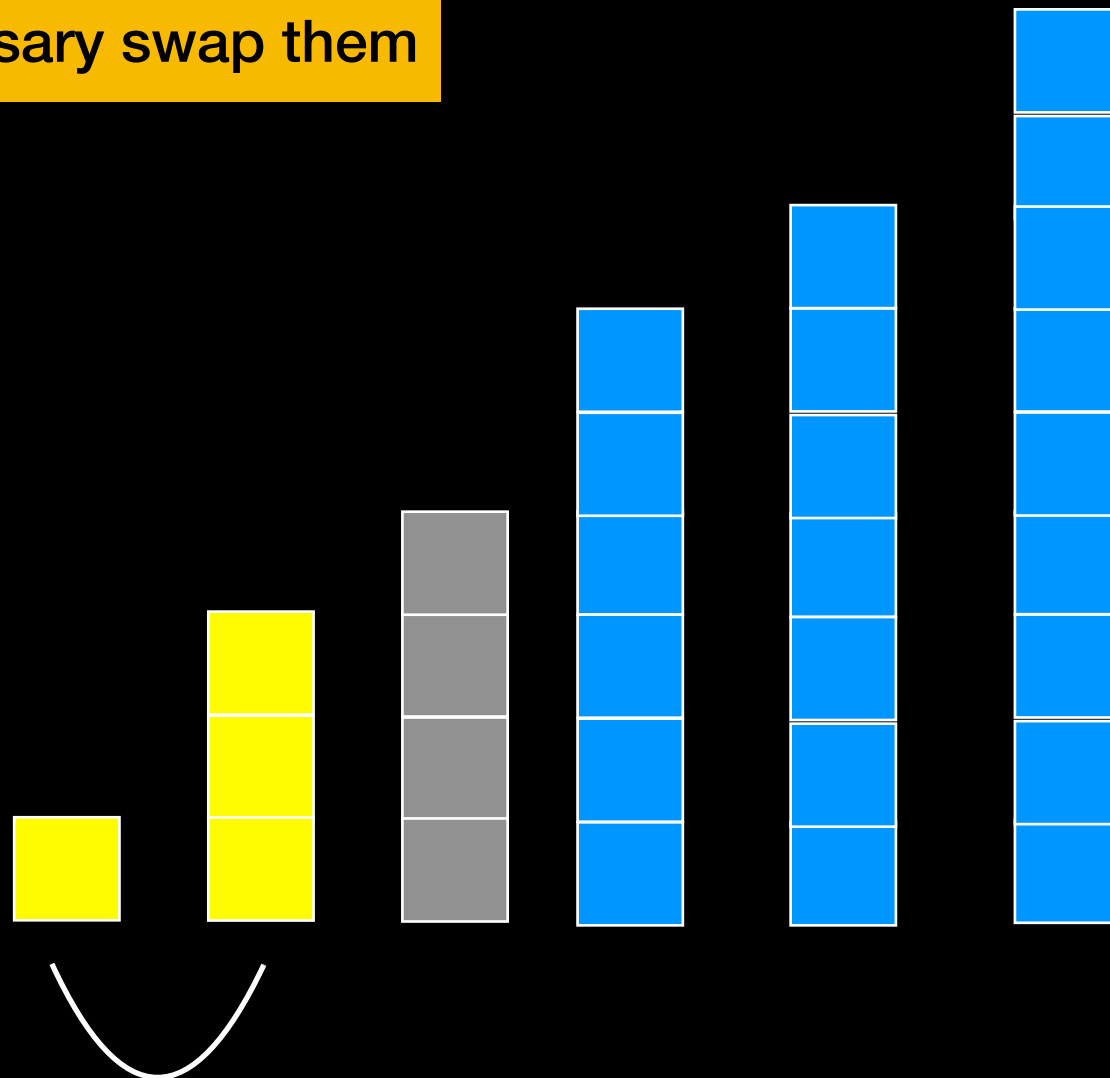
Bubble Sort



Compare adjacent elements
and if necessary swap them



4th Pass



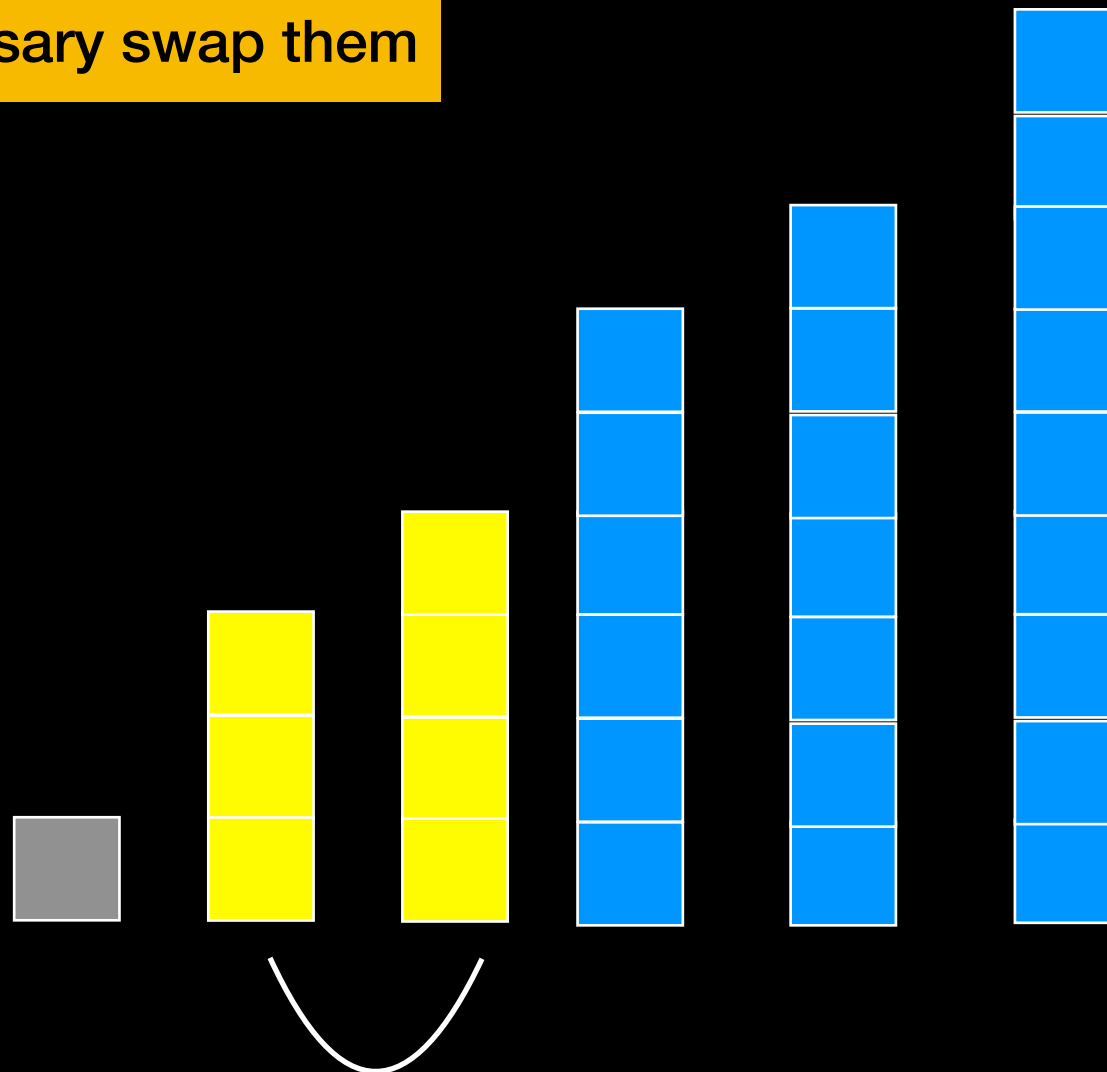
Bubble Sort



Compare adjacent elements
and if necessary swap them



4th Pass

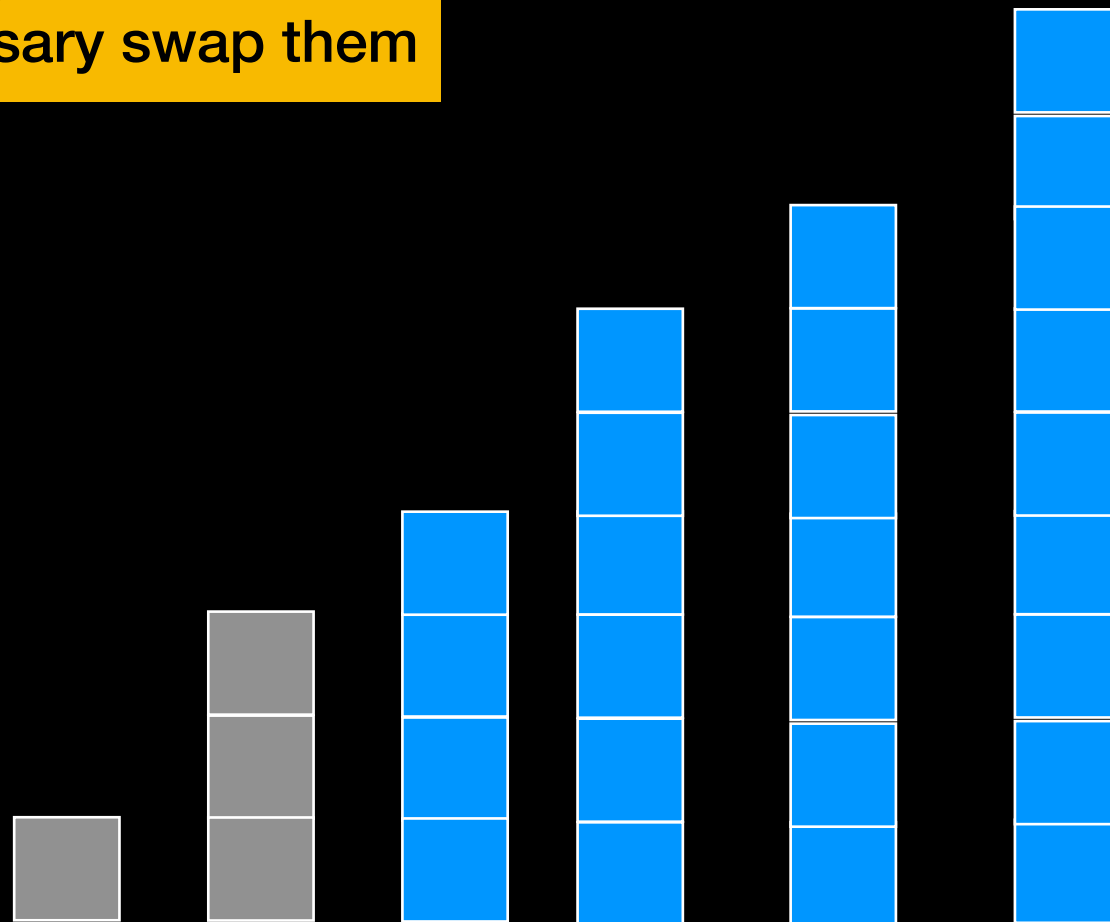


Bubble Sort

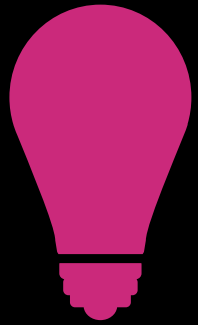


Compare adjacent elements
and if necessary swap them

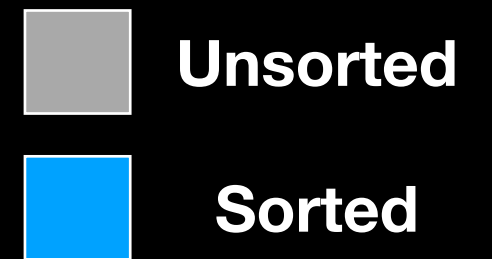
5th Pass:
Sort **n-4**



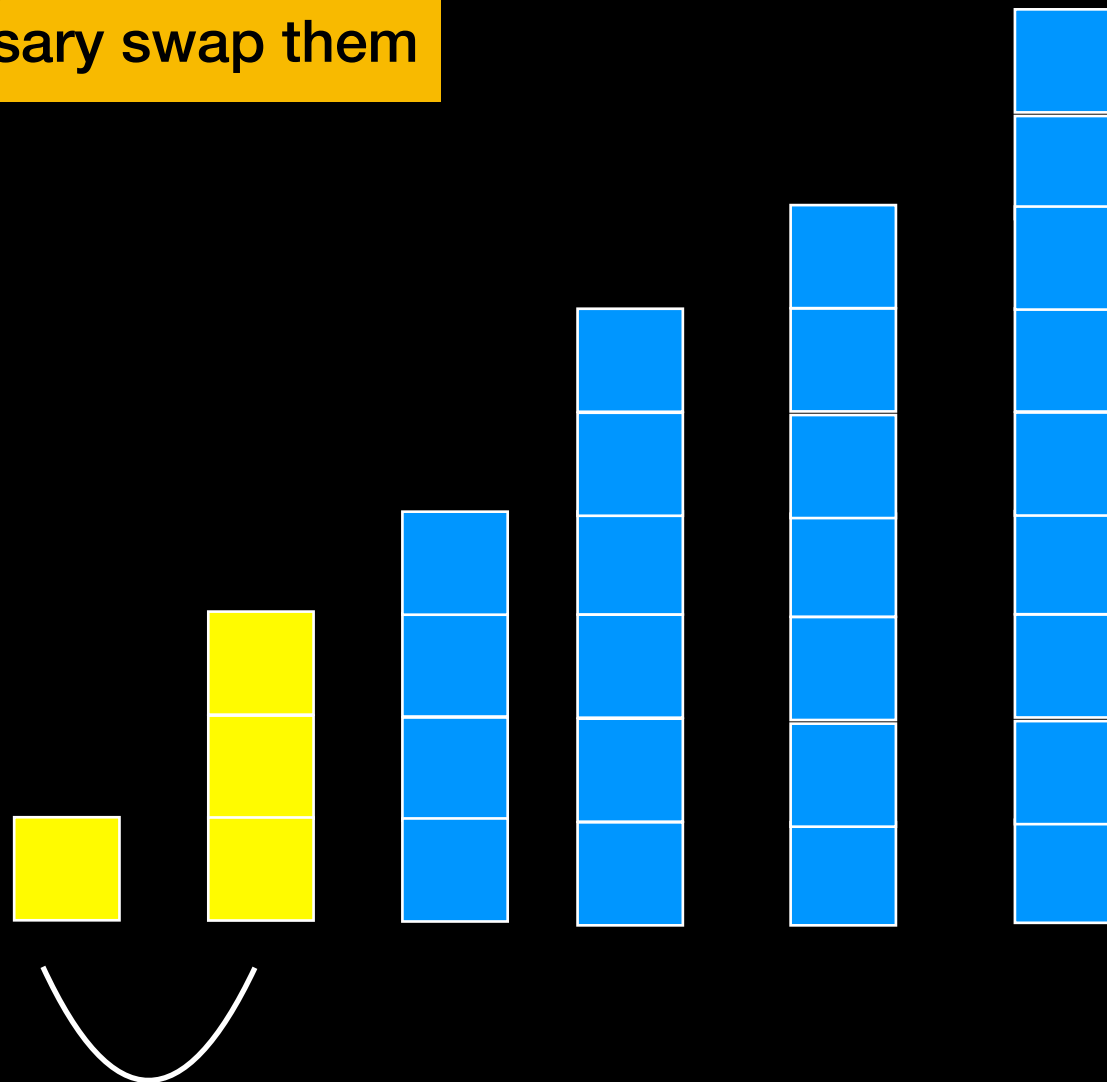
Bubble Sort



Compare adjacent elements
and if necessary swap them



5th Pass



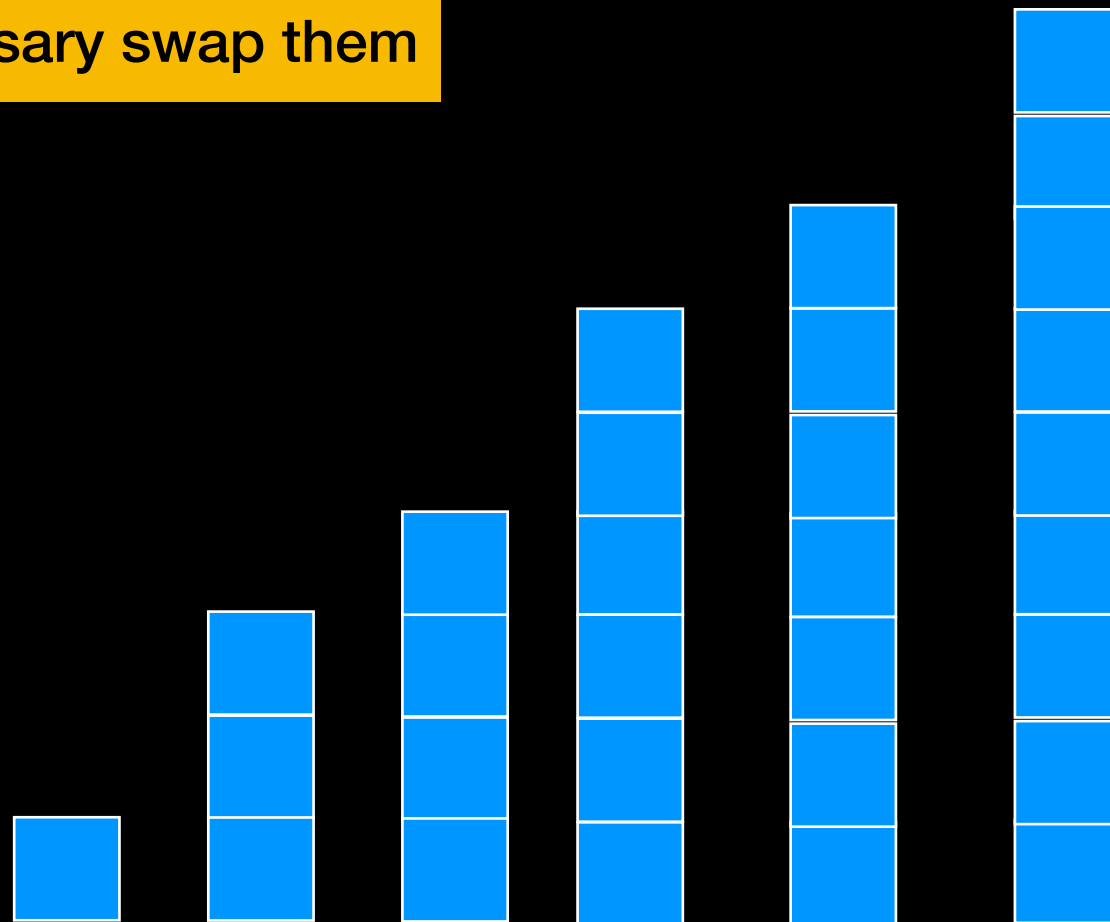
Bubble Sort



Compare adjacent elements
and if necessary swap them

Unsorted
Sorted

Done!



Bubble Sort Analysis

How much work?

First pass: $n-1$ comparisons and at most $n-1$ swaps

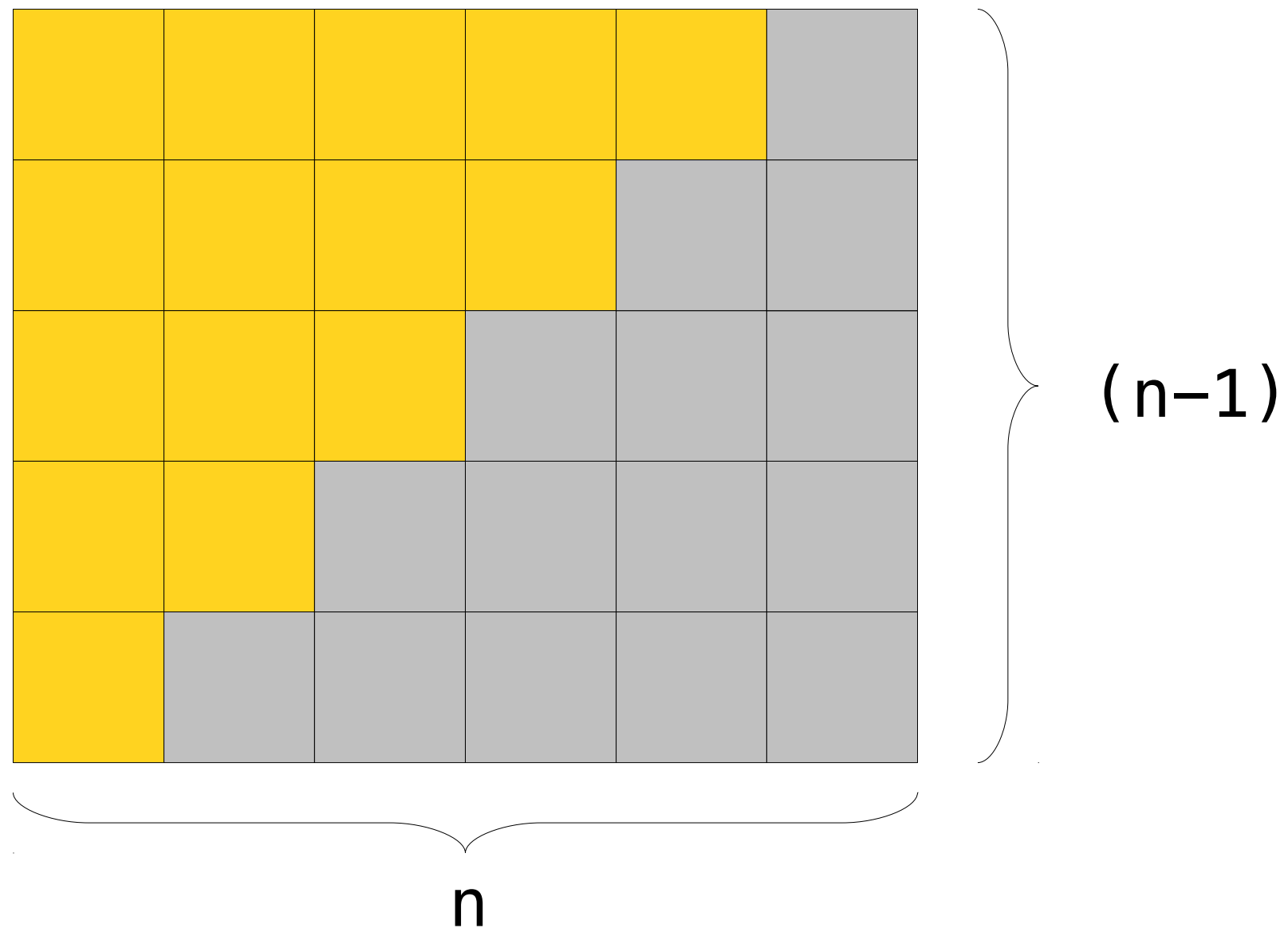
Second pass: $n-2$ comparisons and at most $n-2$ swaps

Third pass: $n-3$ comparisons and at most $n-3$ swaps

...

Total work: $(n-1) + (n-2) + \dots + 1$

$$(n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2$$



Bubble Sort Analysis

$$T(n) = n(n-1) / 2 \text{ comparisons} + n(n-1) / 2 \text{ swaps} = O(\text{ })?$$

A swap is usually more than one operation but this simplification does not change the analysis

$$T(n) = 2(n(n-1) / 2) = O(\text{ })?$$

Bubble Sort Analysis

$$T(n) = n(n-1) / 2 \text{ comparisons} + n(n-1) / 2 \text{ swaps} = O(\text{ })?$$

A swap is usually more than one operation but this simplification does not change the analysis

$$T(n) = 2(n(n-1) / 2) = O(\text{ })?$$

$$T(n) = 2((n^2-n) / 2) = O(\text{ })?$$

Bubble Sort Analysis

$$T(n) = n(n-1) / 2 \text{ comparisons} + n(n-1) / 2 \text{ swaps} = O()?$$

A swap is usually more than one operation but this simplification does not change the analysis

$$T(n) = 2(n(n-1) / 2) = O()?$$

$$T(n) = 2((n^2-n) / 2) = O()?$$

$$T(n) = n^2 - n = O()?$$

Ignore non-dominant terms

Bubble Sort Analysis

$$T(n) = n(n-1) / 2 \text{ comparisons} + n(n-1) / 2 \text{ swaps} = O()?$$

A swap is usually more than one operation but this simplification does not change the analysis

$$T(n) = 2(n(n-1) / 2) = O()?$$

$$T(n) = 2((n^2-n) / 2) = O()?$$

$$T(n) = n^2 - n = O()?$$

Bubble Sort run time is $O(n^2)$

Optimize!

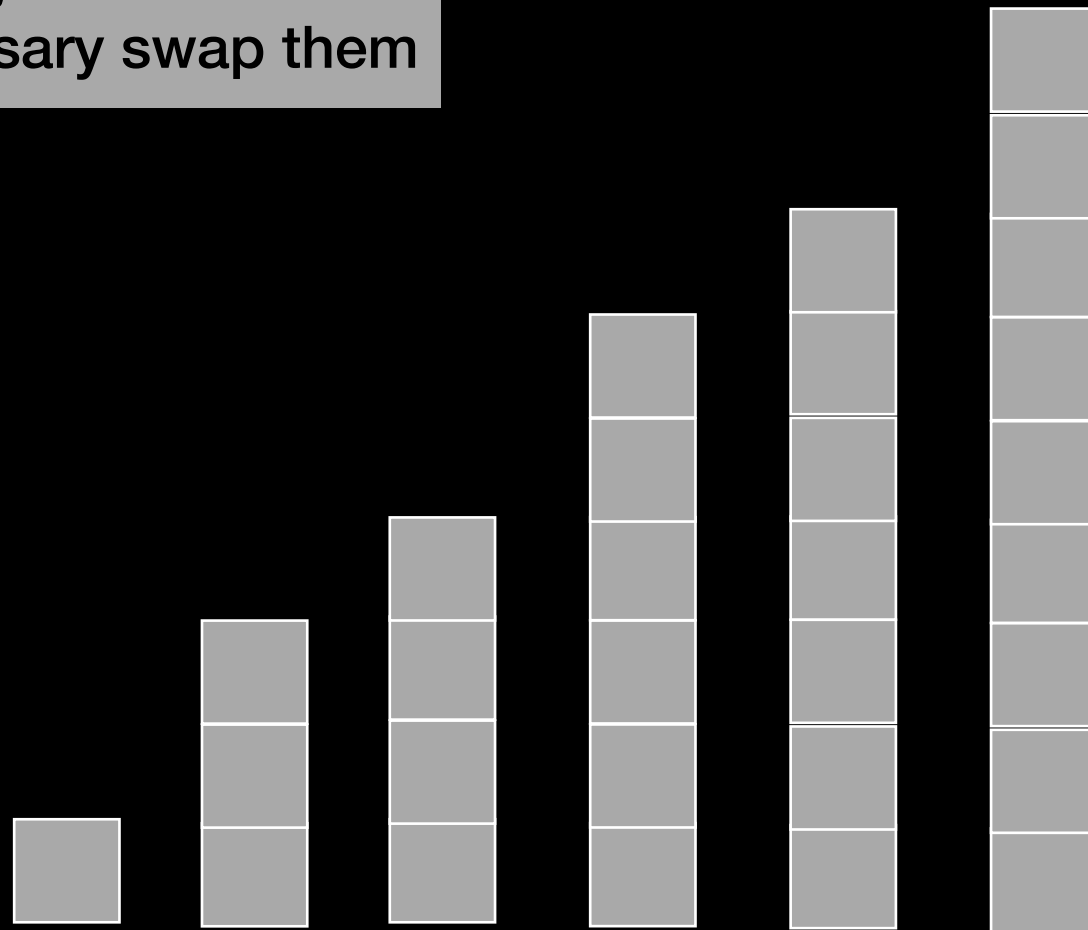
Easy to check:

if there are no swaps in any given pass
stop because it is sorted

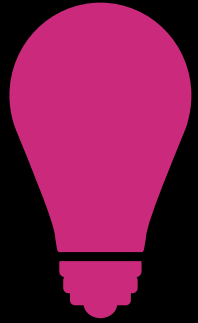
Bubble Sort



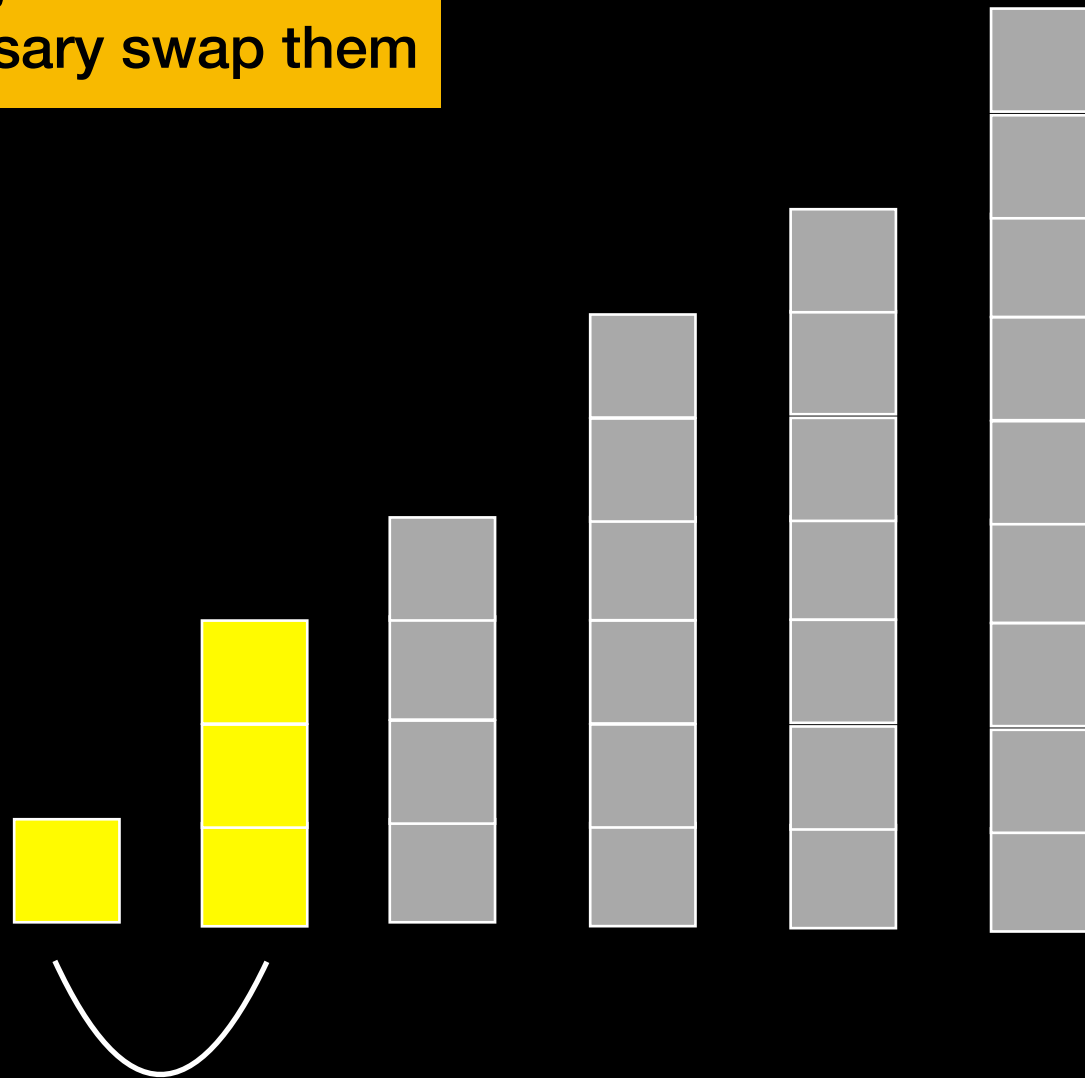
Compare adjacent elements
and if necessary swap them



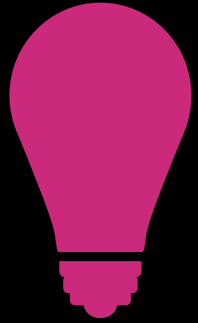
Bubble Sort



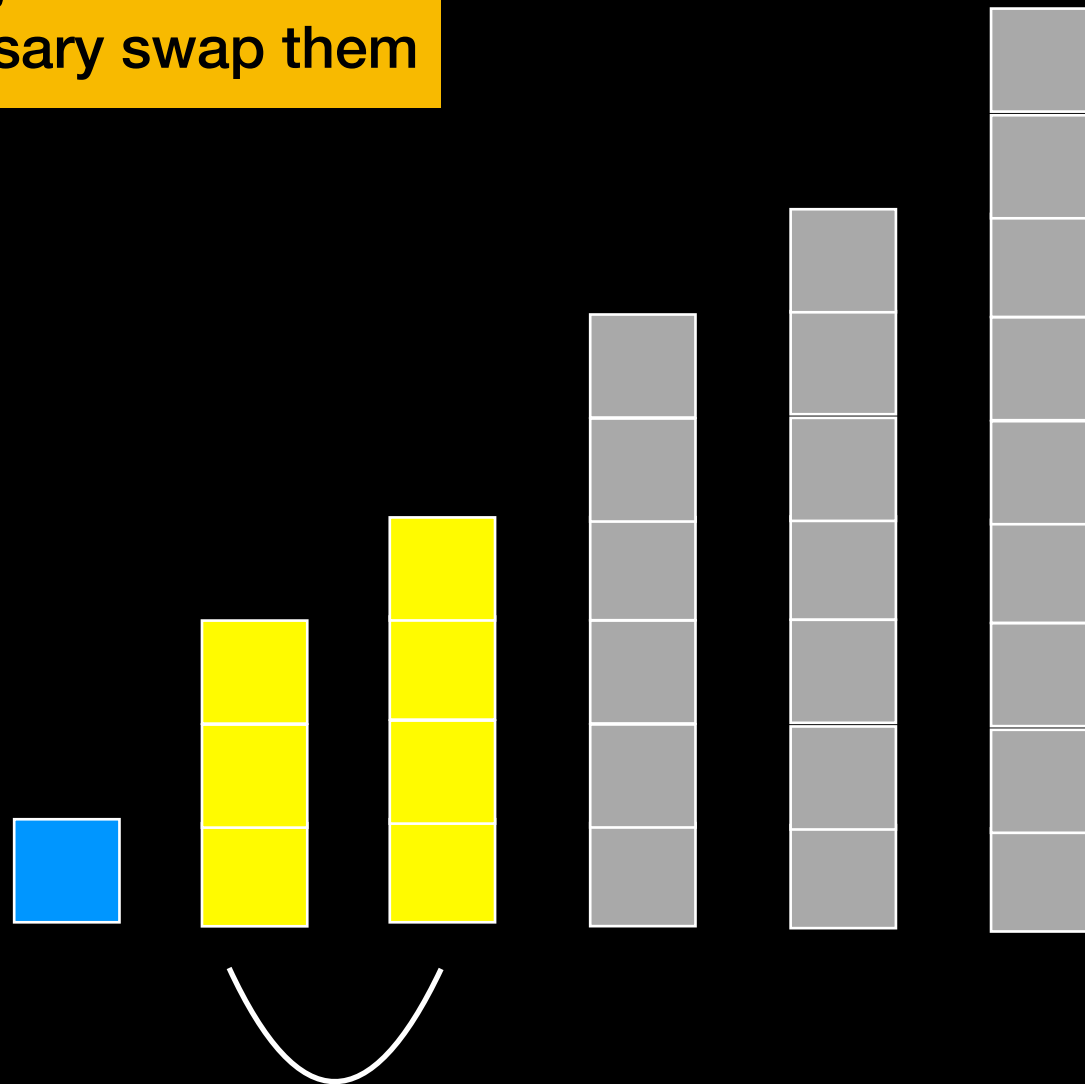
Compare adjacent elements
and if necessary swap them



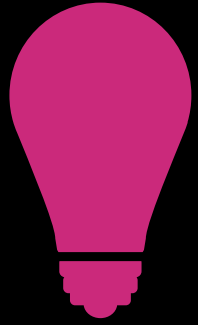
Bubble Sort



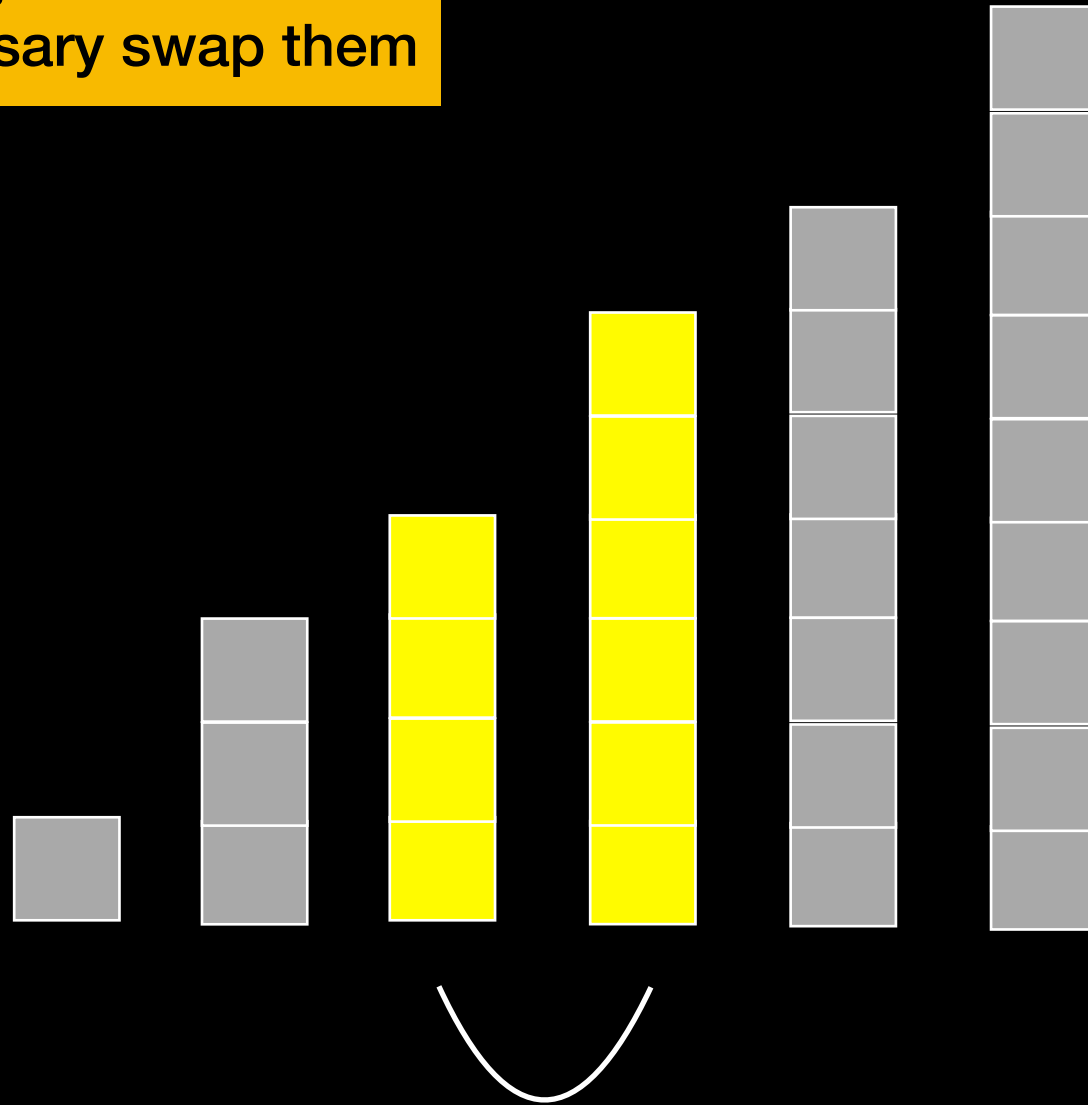
Compare adjacent elements
and if necessary swap them



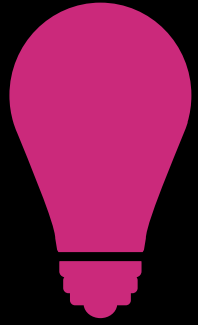
Bubble Sort



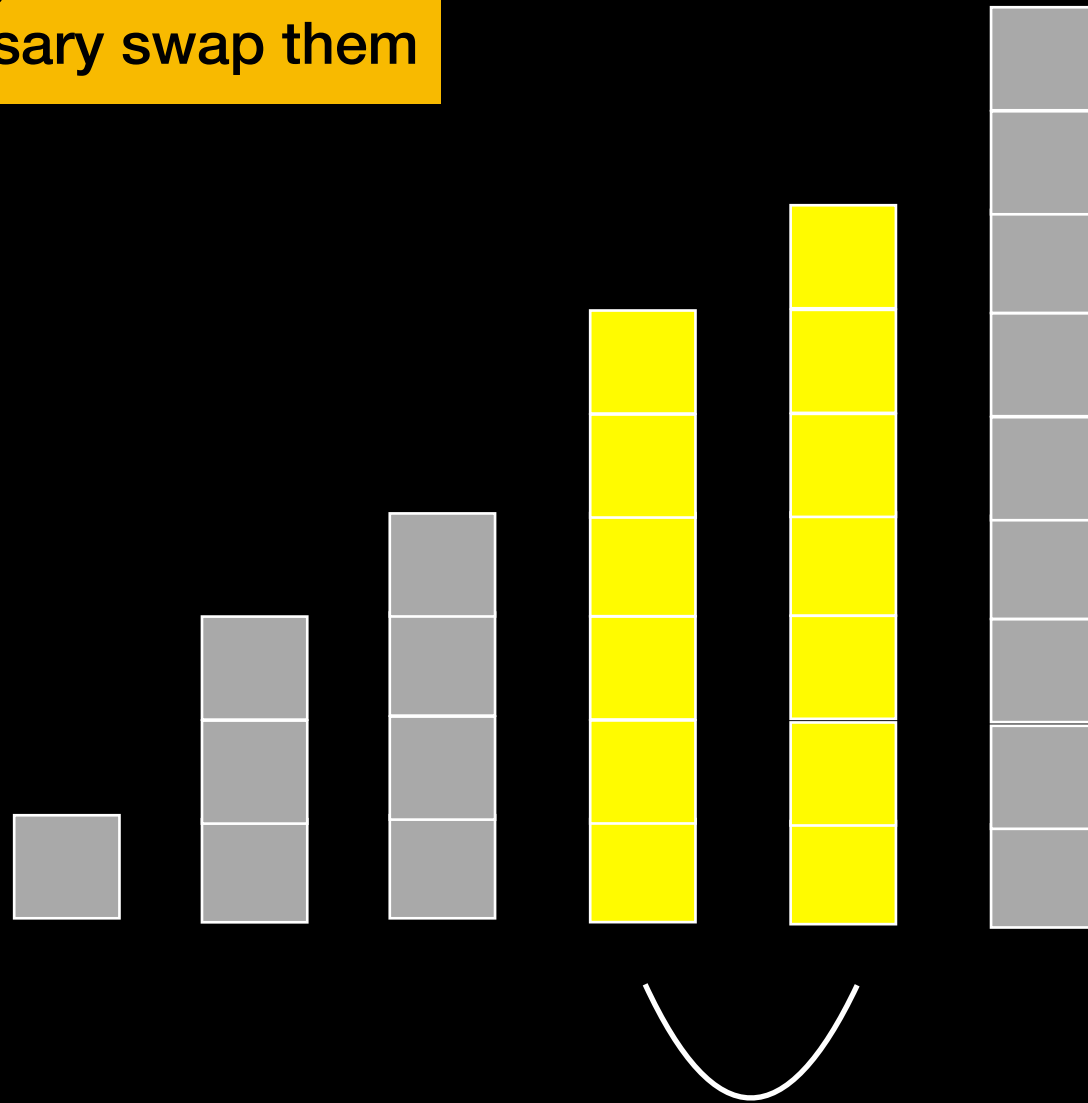
Compare adjacent elements
and if necessary swap them



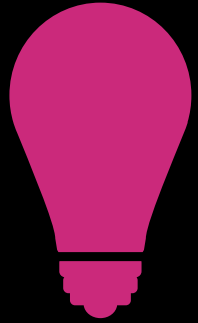
Bubble Sort



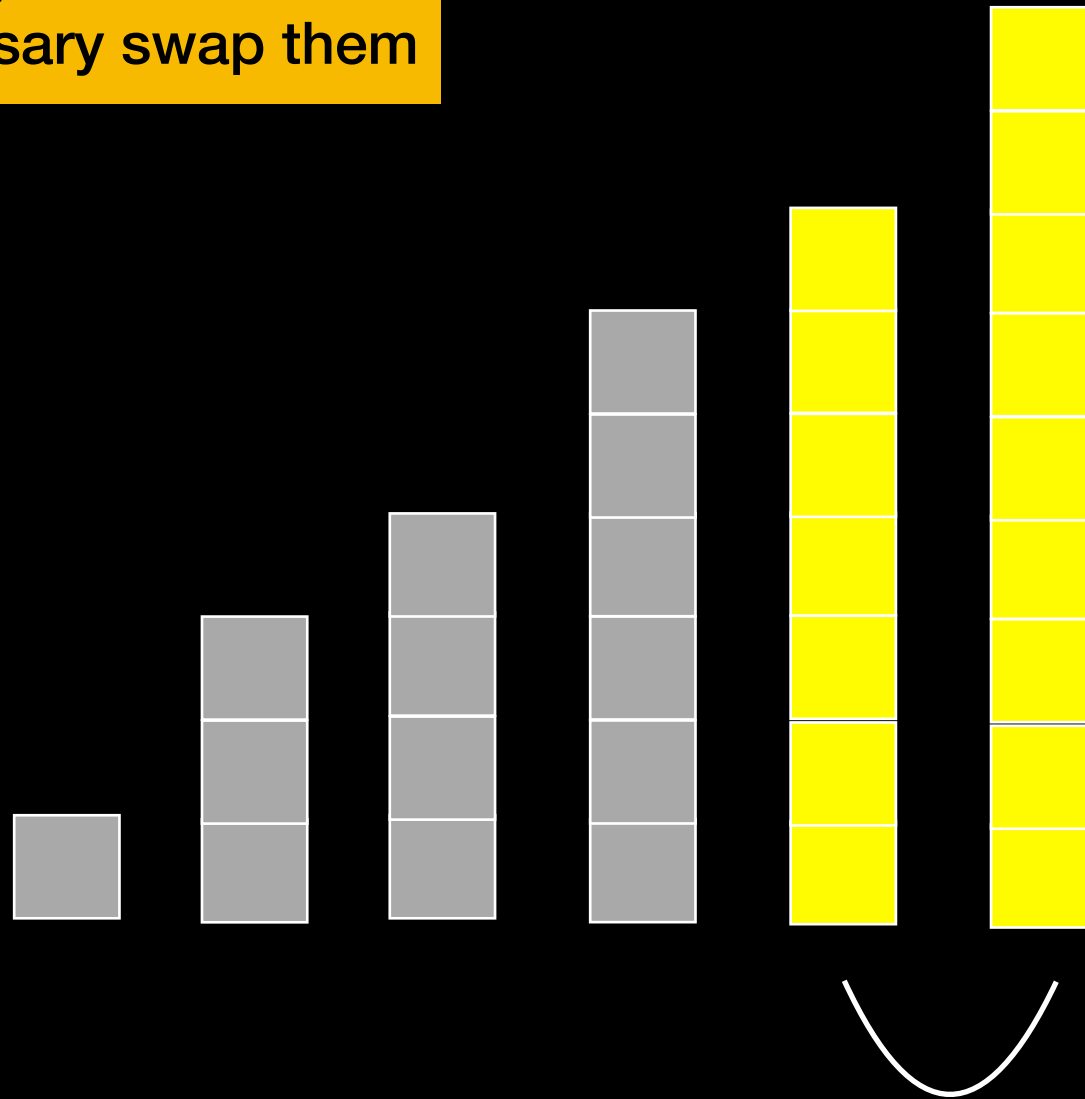
Compare adjacent elements
and if necessary swap them



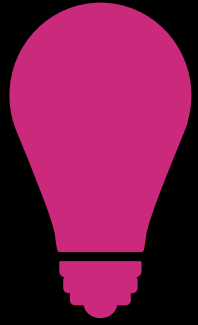
Bubble Sort



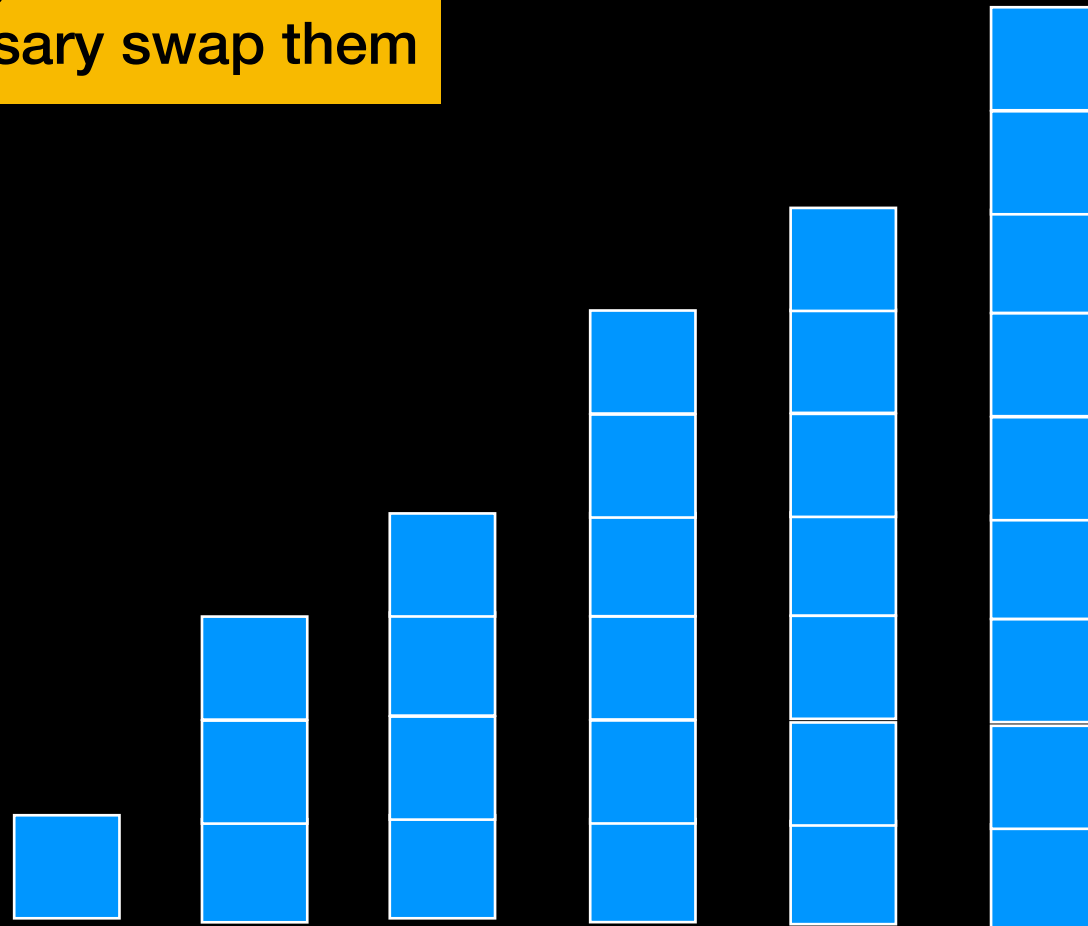
Compare adjacent elements
and if necessary swap them



Bubble Sort



Compare adjacent elements
and if necessary swap them



```

template<typename ItemType>
void bubbleSort(ItemType the_array[], size_t size)
{
    bool swapped = true; // Assume unsorted
    int pass = 1;
    while (swapped && (pass < size))
    {
        // At this point, if pass > 1 the_array[size+1-pass ... size-1]
        // is sorted and all of its entries are > the entries in
        // the_array[0 ... size-pass]
        swapped = false;
        for (int index = 0; index < size - pass; index++)
        {
            // At this point, all entries in the_array[0 ... index-1]
            // are <= the_array[index]
            if (the_array[index] > the_array[index+1])
            {
                swap(the_array[index], the_array[index+1]); //swap
                swapped = true; // indicates array not yet sorted
            } // end if
        } // end for
        //Assertion:the_array[0 ... size-pass-1]<the_array[size-pass]
        pass++;
    } // end while
} // end bubbleSort

```

```

template<typename ItemType>
void bubbleSort(ItemType the_array[], size_t size)
{
    bool swapped = true; // Assume unsorted
    int pass = 1;
    Pass while (swapped && (pass < size))
    O(n) {
        // At this point, if pass > 1 the_array[size+1-pass ... size-1]
        // is sorted and all of its entries are > the entries in
        // the_array[0 ... size-pass]
        swapped = false;
        O(n) for (int index = 0; index < size - pass; index++)
        {
            // At this point, all entries in the_array[0 ... index-1]
            // are <= the_array[index]
            if (the_array[index] > the_array[index+1])
            {
                swap(the_array[index], the_array[index+1]); //swap
                swapped = true; // indicates array not yet sorted
            } // end if
        } // end for
        //Assertion:the_array[0 ... size-pass-1]<the_array[size-pass]
        pass++;
    } // end while
} // end bubbleSort

```

O(n²)

Bubble Sort Analysis

Execution time DOES depend on initial arrangement of data

Worst case: $O(n^2)$ comparisons and data moves

Best case: $O(n)$ comparisons and data moves

Stable

If array is already sorted bubble sort will stop after first pass and no swaps => good choice for **small n** and data likely **somewhat sorted**

Raise your hand if you had
Bubble Sort

<https://www.youtube.com/watch?v=lyZQPjUT5B4>




Insertion Sort

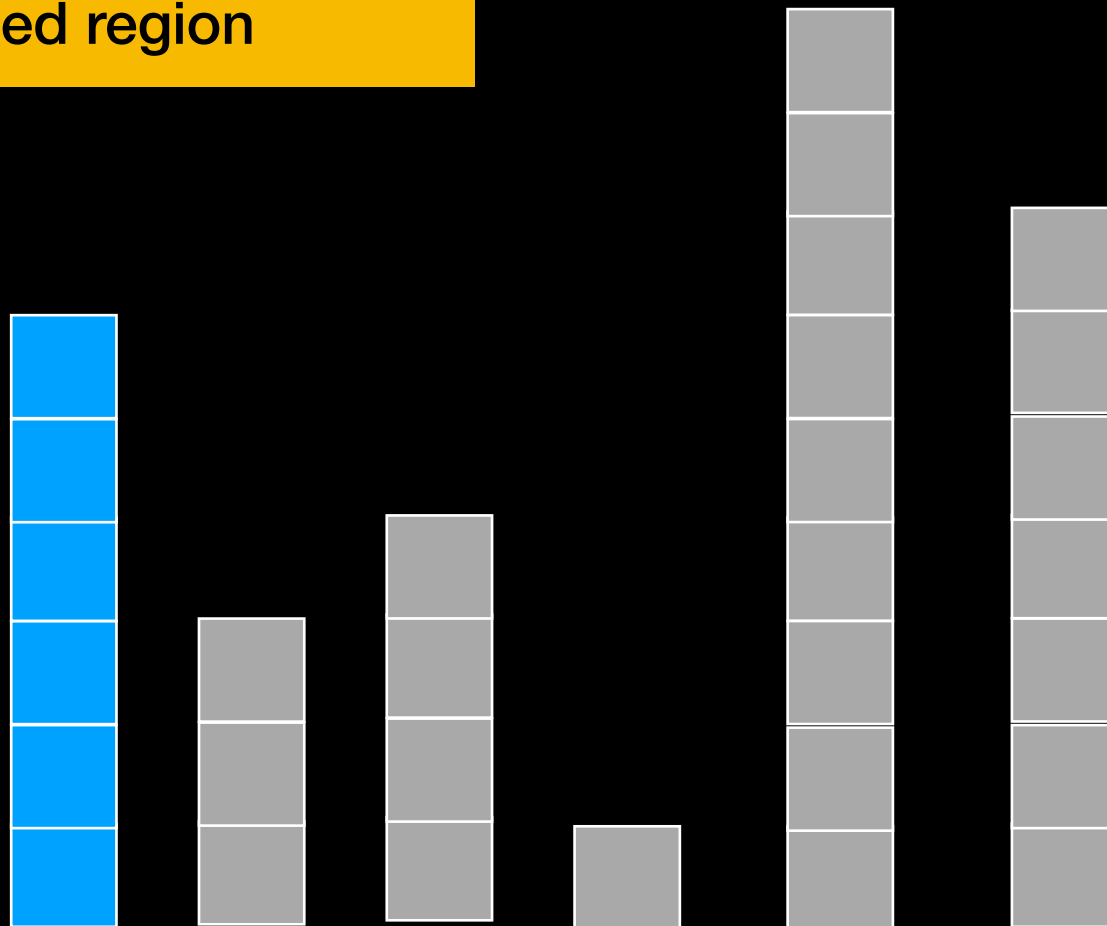
Insertion Sort



Pick first element in unsorted region and put it in right place in sorted region

 Unsorted
 Sorted

1st Pass



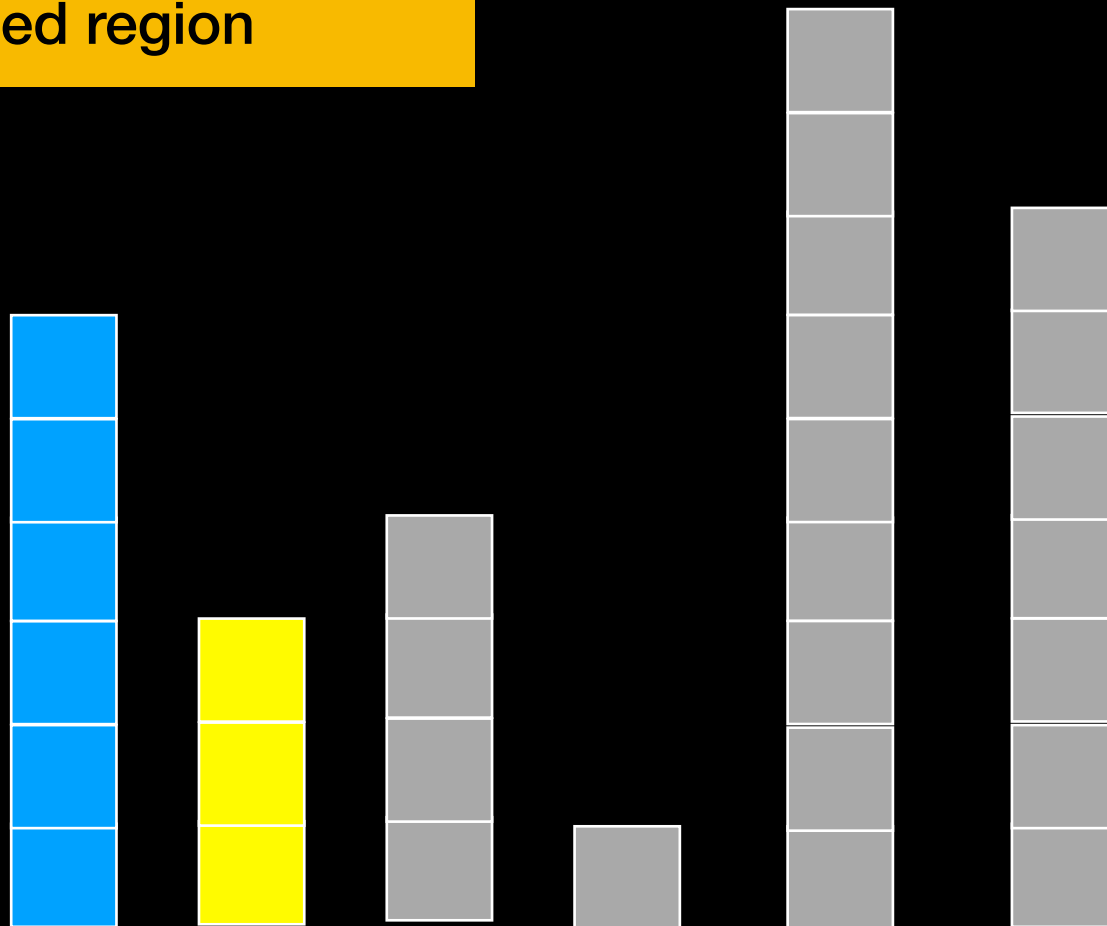
Insertion Sort



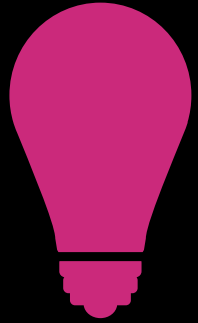
Pick first element in unsorted region and put it in right place in sorted region



1st Pass



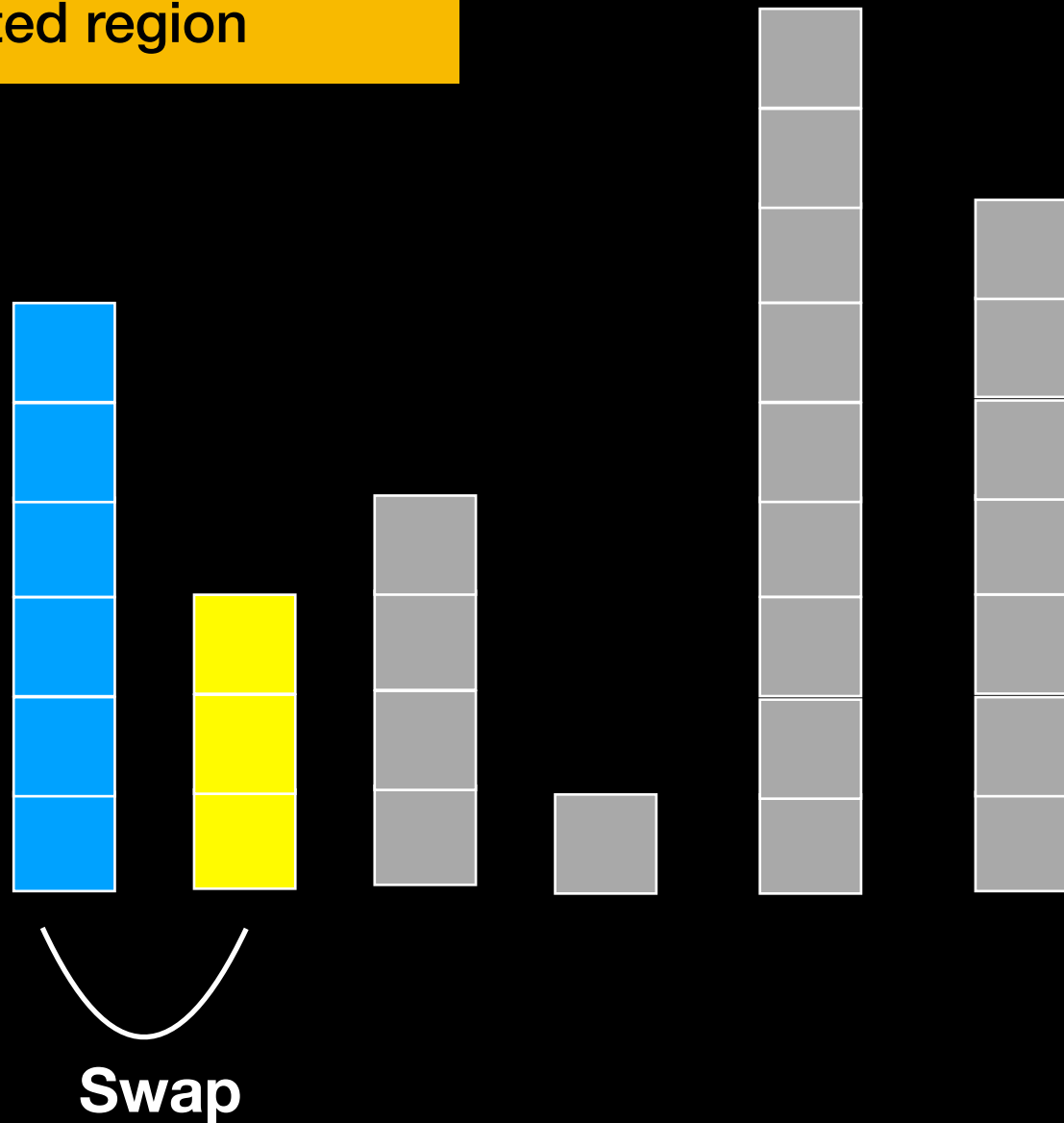
Insertion Sort



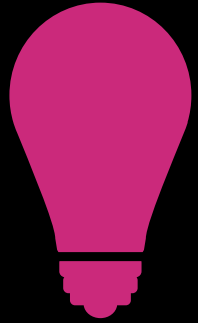
Pick first element in unsorted region and put it in right place in sorted region



1st Pass



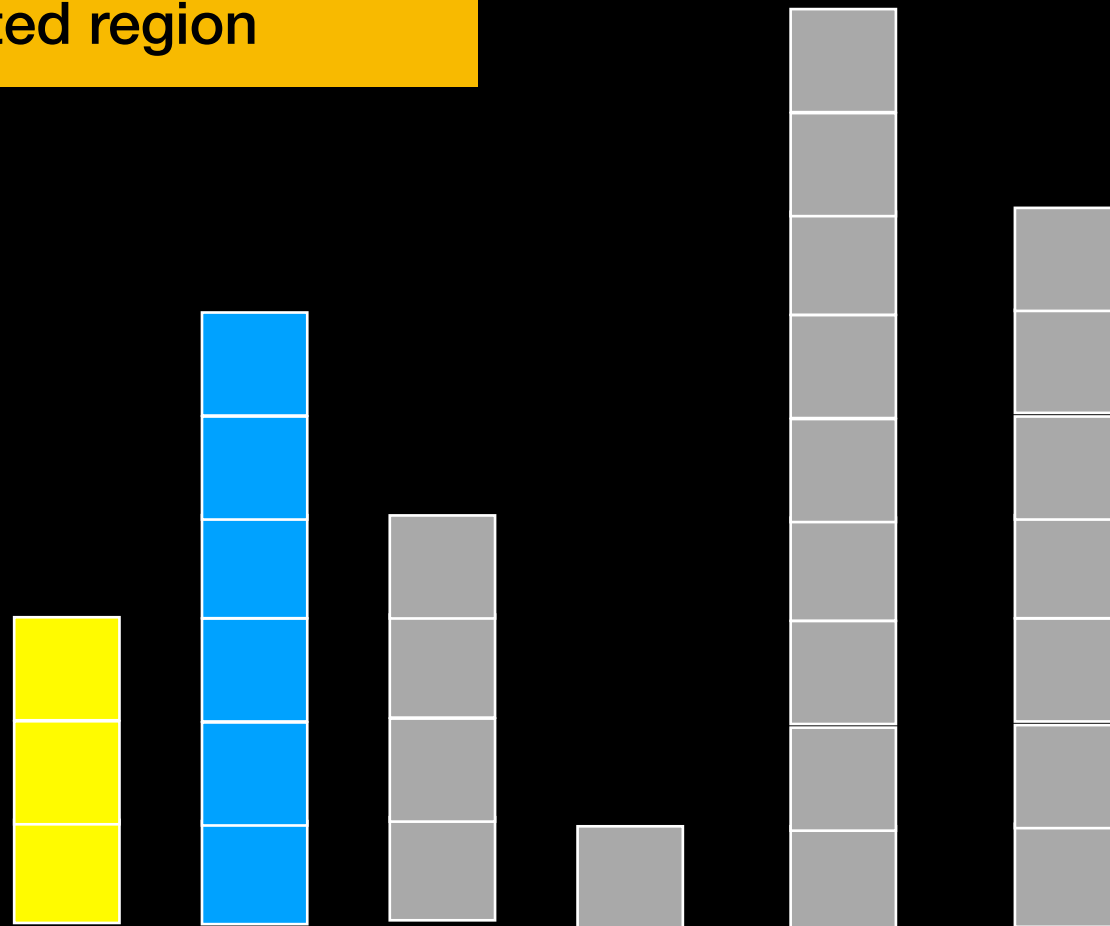
Insertion Sort



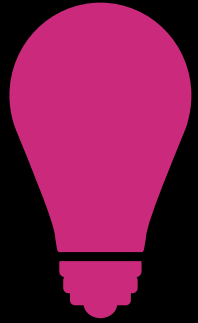
Pick first element in unsorted region and put it in right place in sorted region




1st Pass

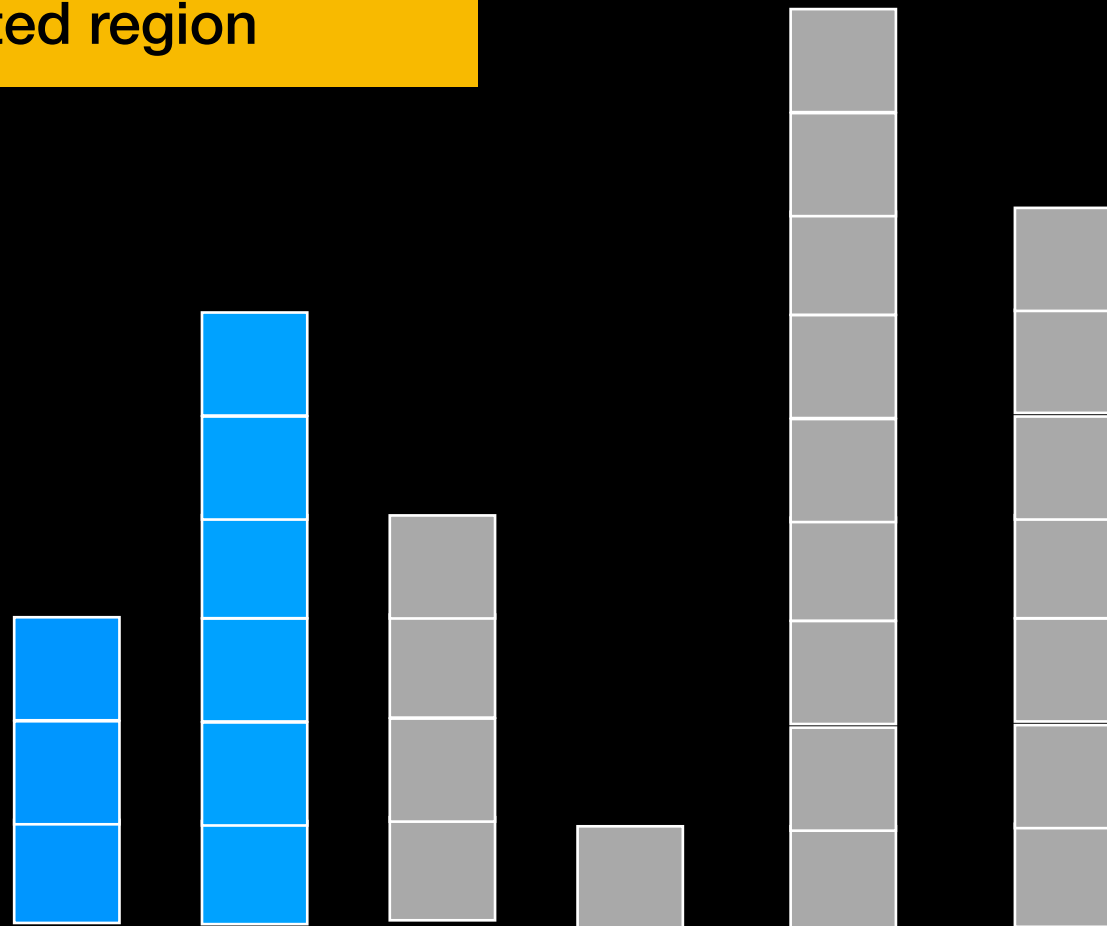


Insertion Sort

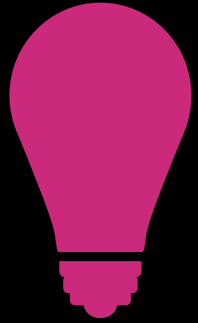


Pick first element in unsorted region and put it in right place in sorted region

 Unsorted
 Sorted



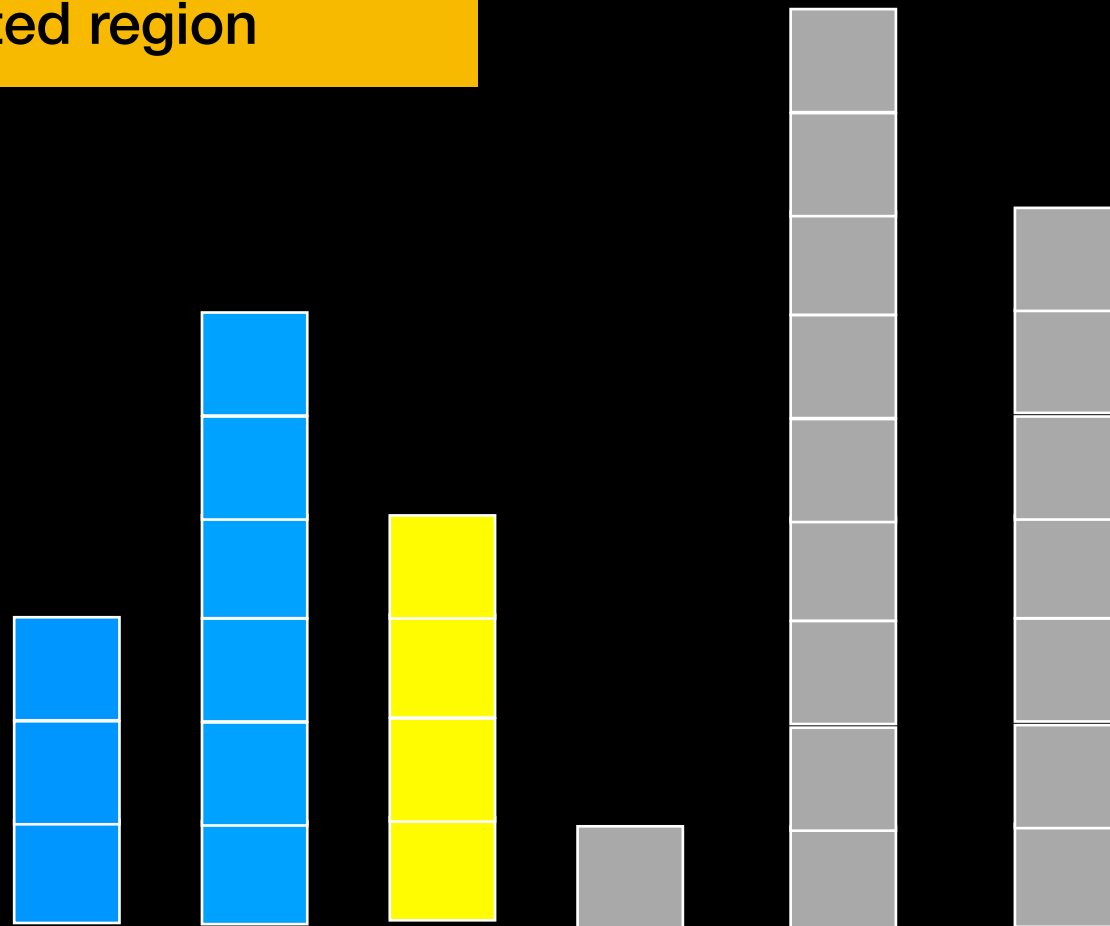
Insertion Sort



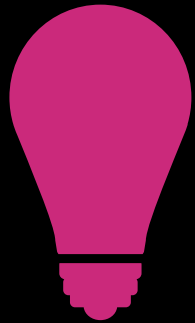
Pick first element in unsorted region and put it in right place in sorted region

 Unsorted
 Sorted

2nd Pass



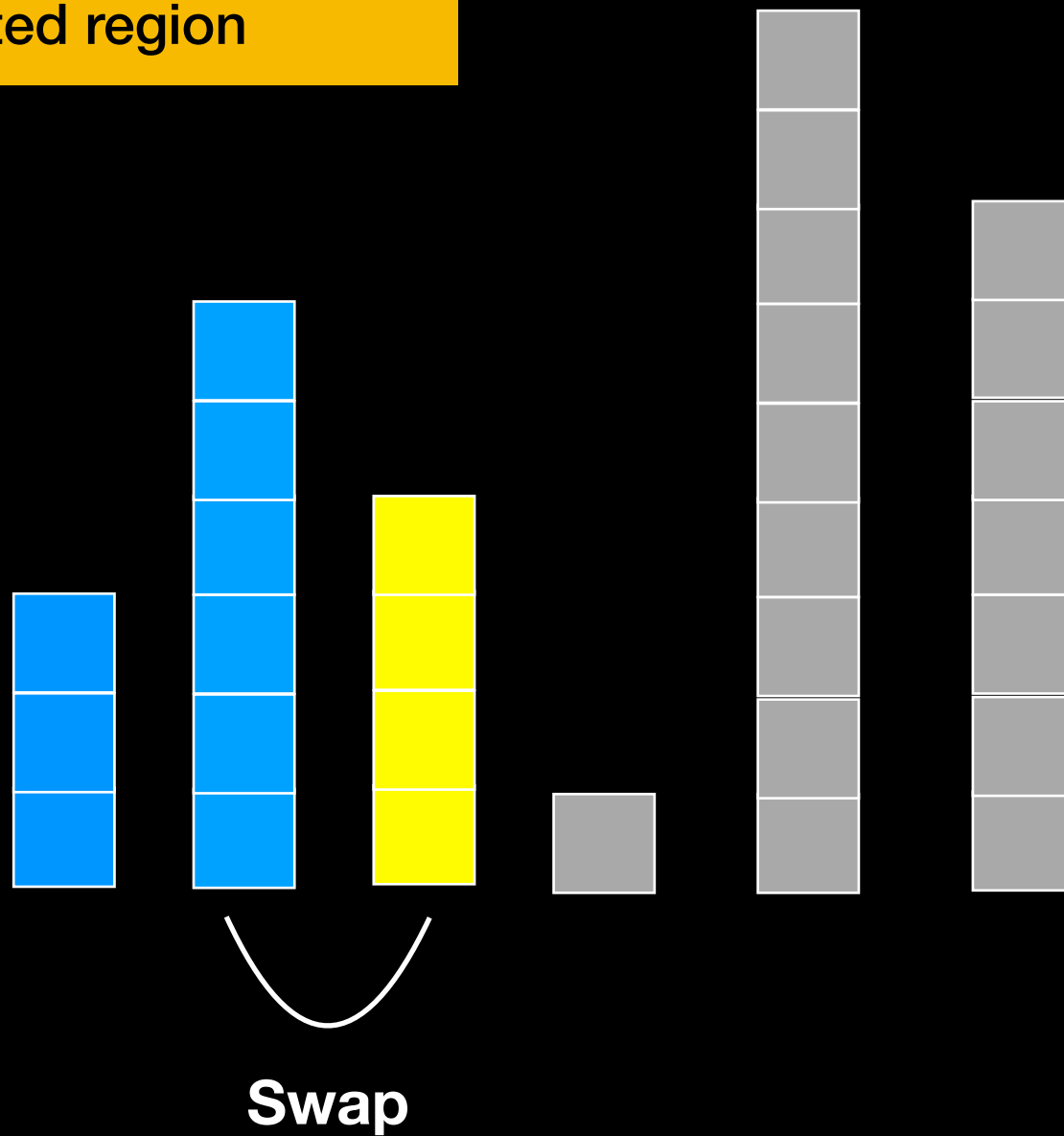
Insertion Sort



Pick first element in unsorted region and put it in right place in sorted region



2nd Pass



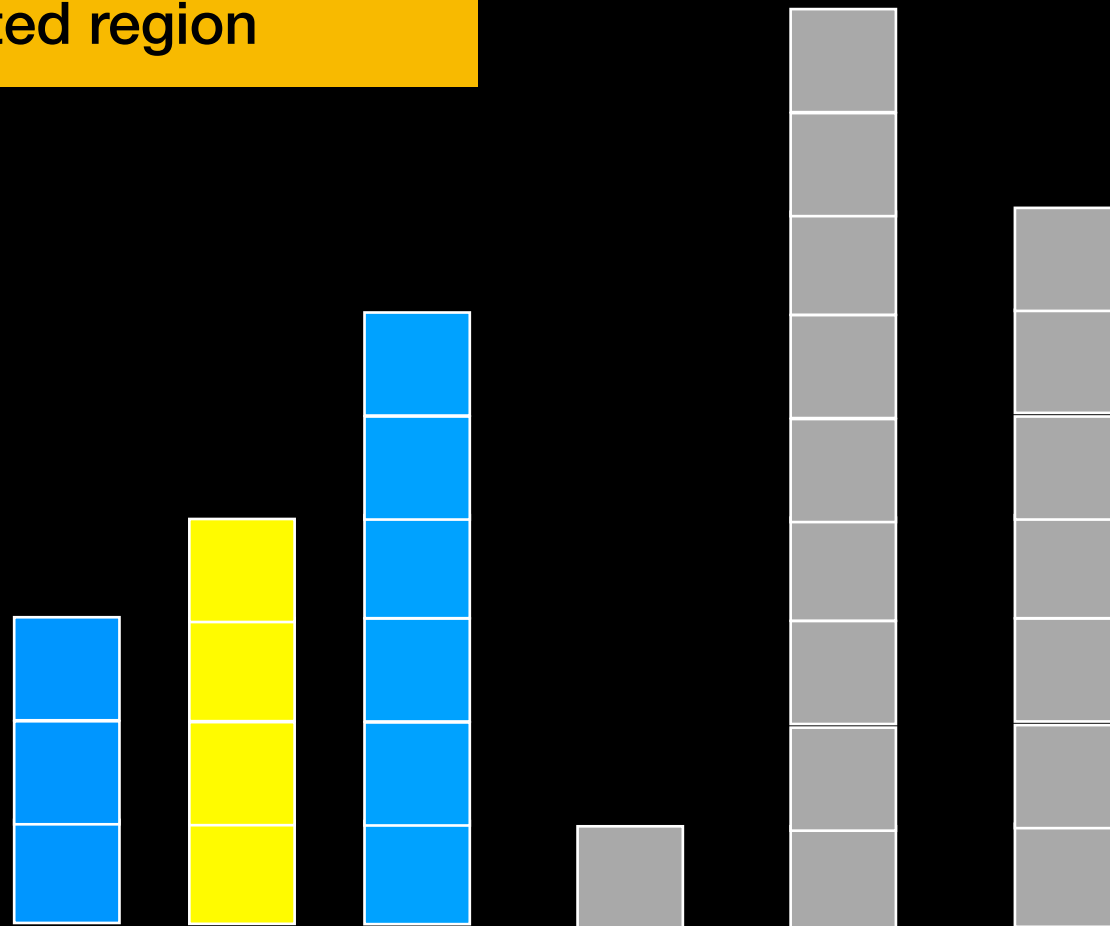
Insertion Sort



Pick first element in unsorted region and put it in right place in sorted region



2nd Pass



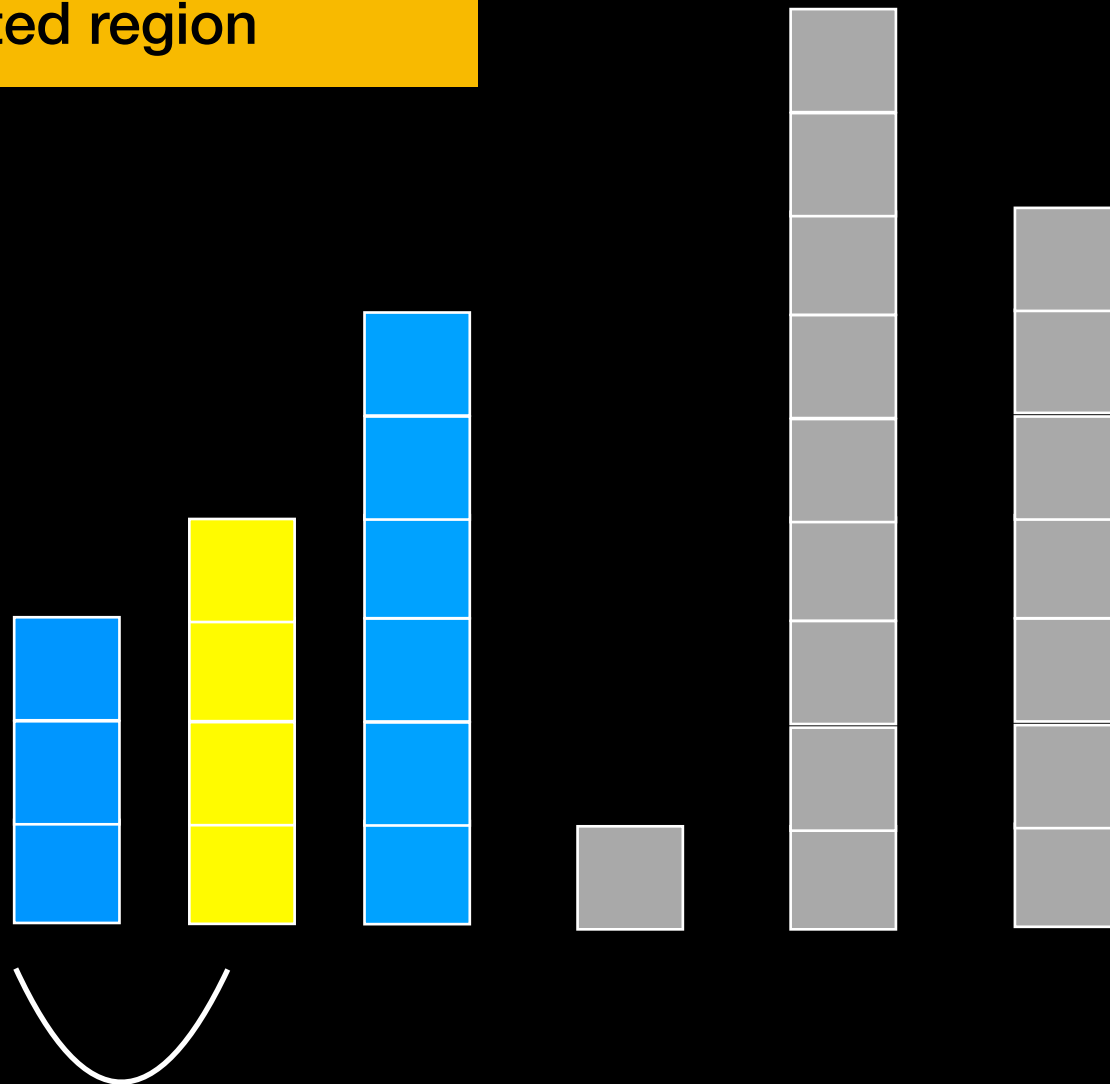
Insertion Sort



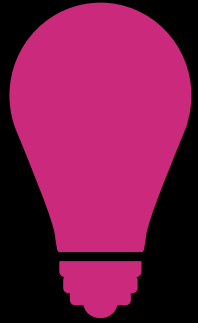
Pick first element in unsorted region and put it in right place in sorted region



2nd Pass

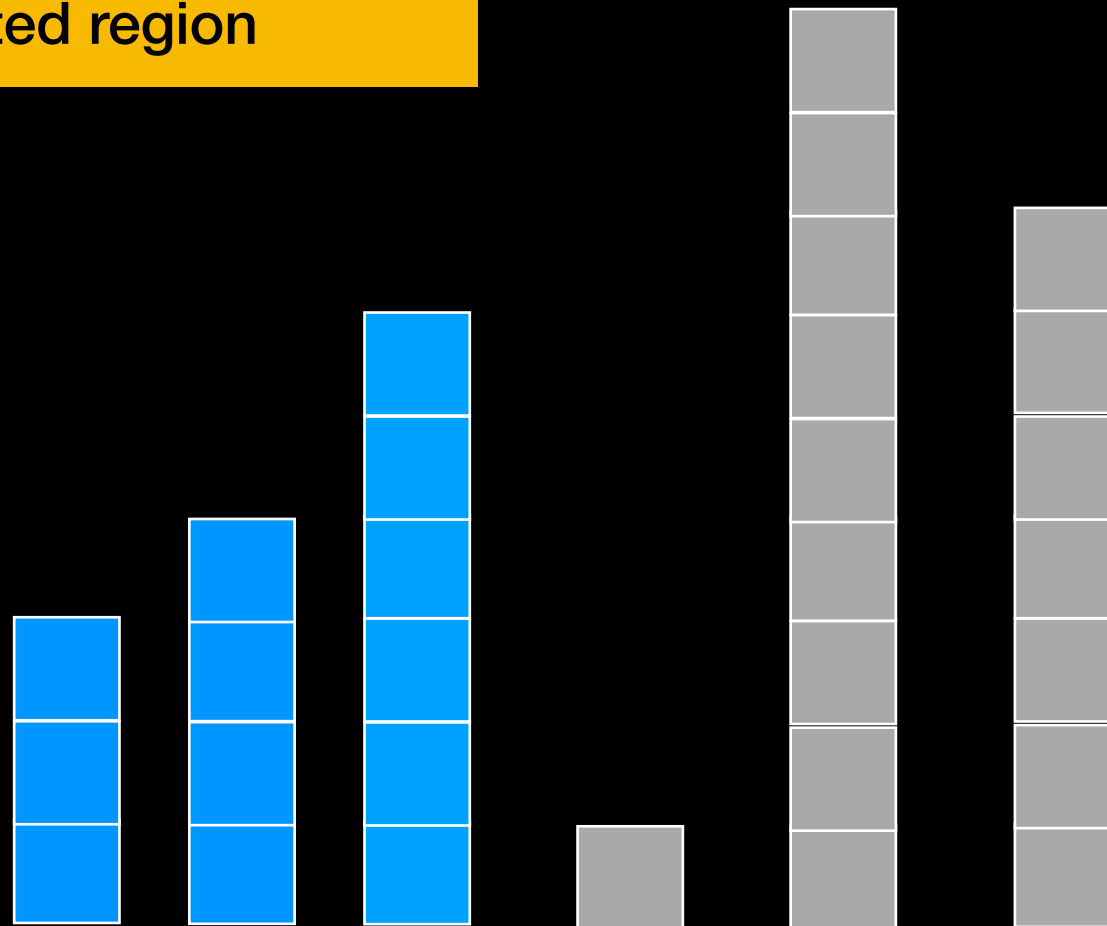


Insertion Sort

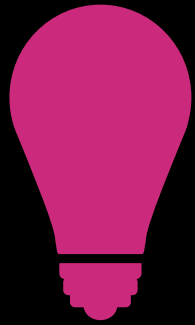


Pick first element in unsorted region and put it in right place in sorted region

 Unsorted
 Sorted



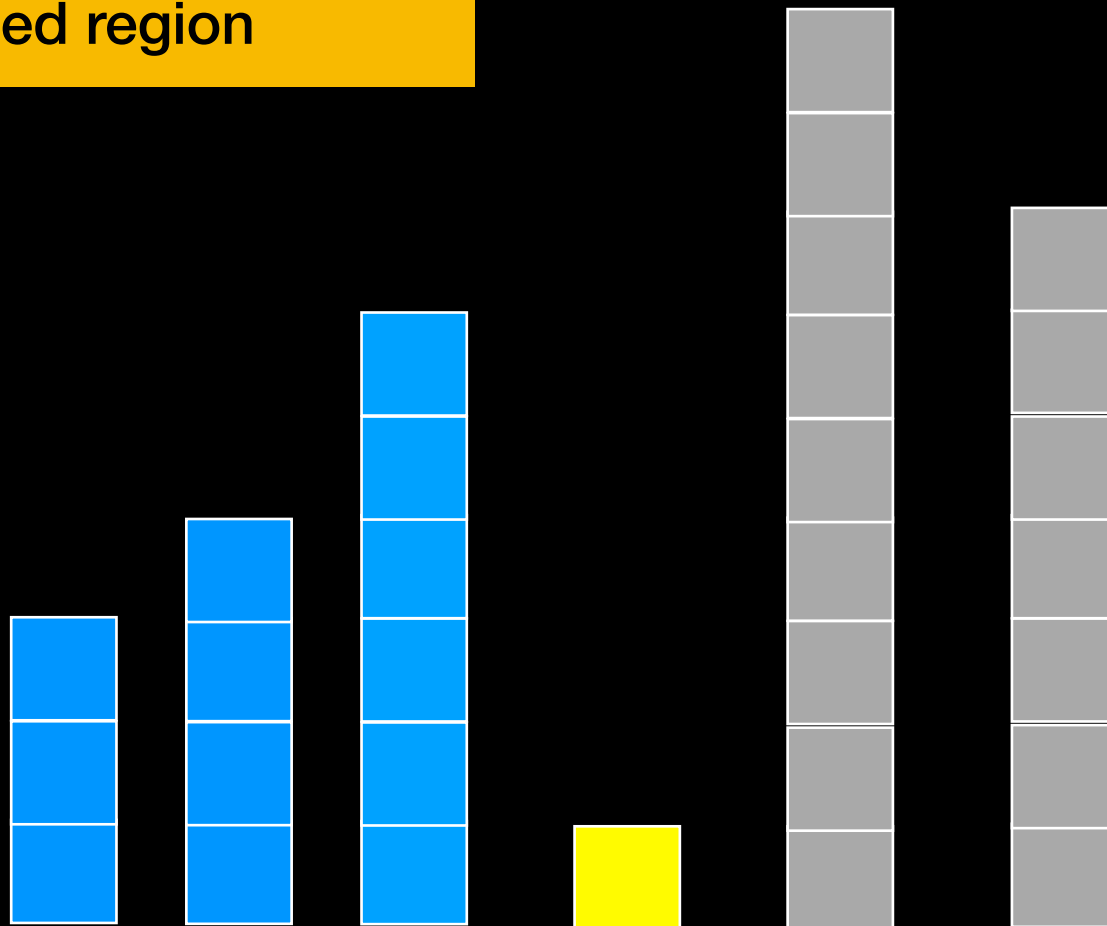
Insertion Sort



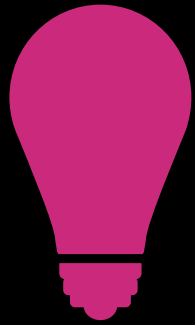
Pick first element in unsorted region and put it in right place in sorted region

Unsorted
Sorted

3rd Pass



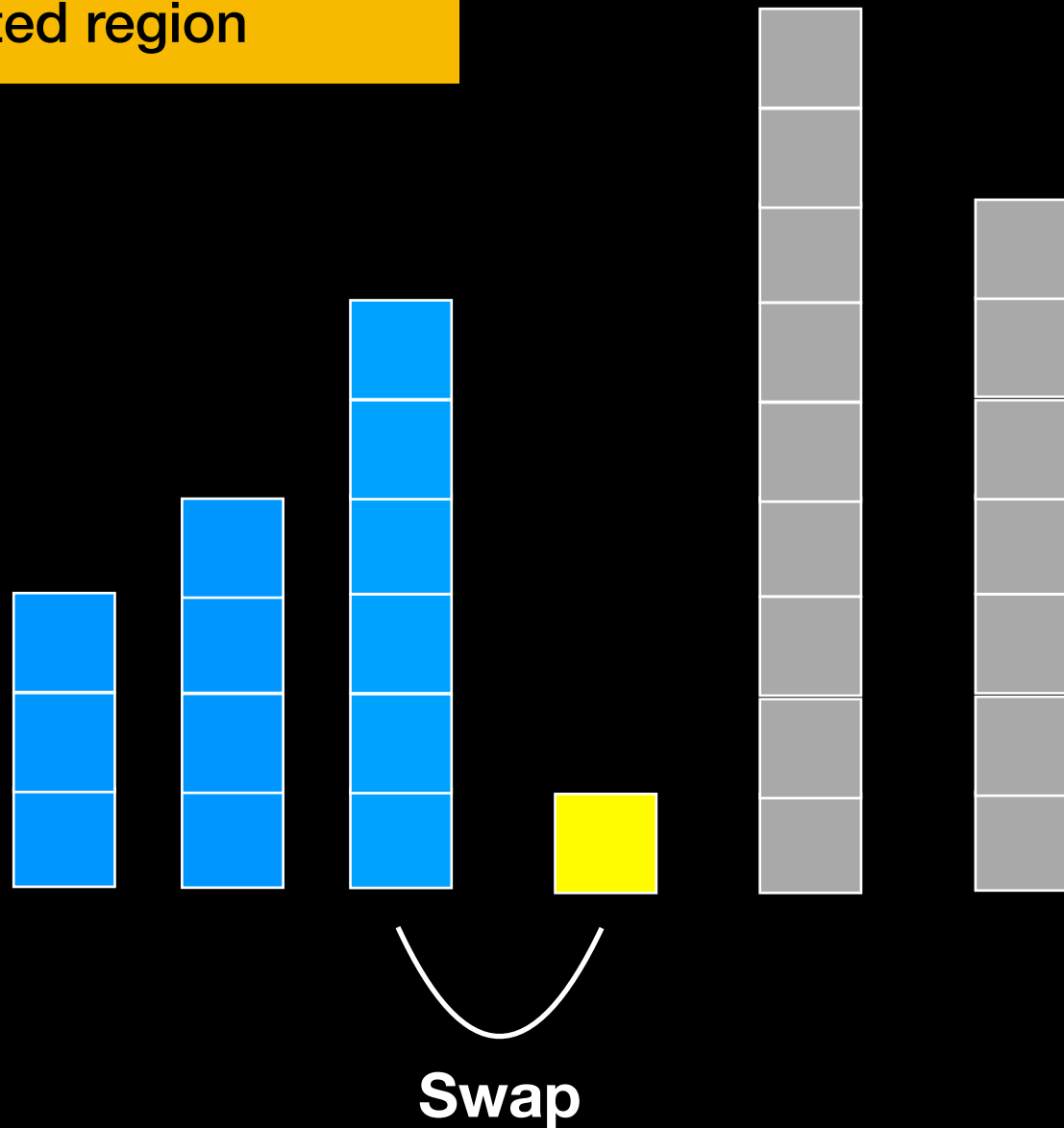
Insertion Sort



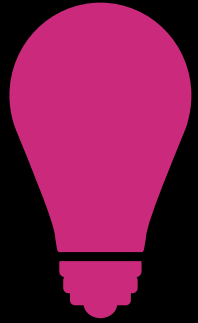
Pick first element in unsorted region and put it in right place in sorted region



3rd Pass



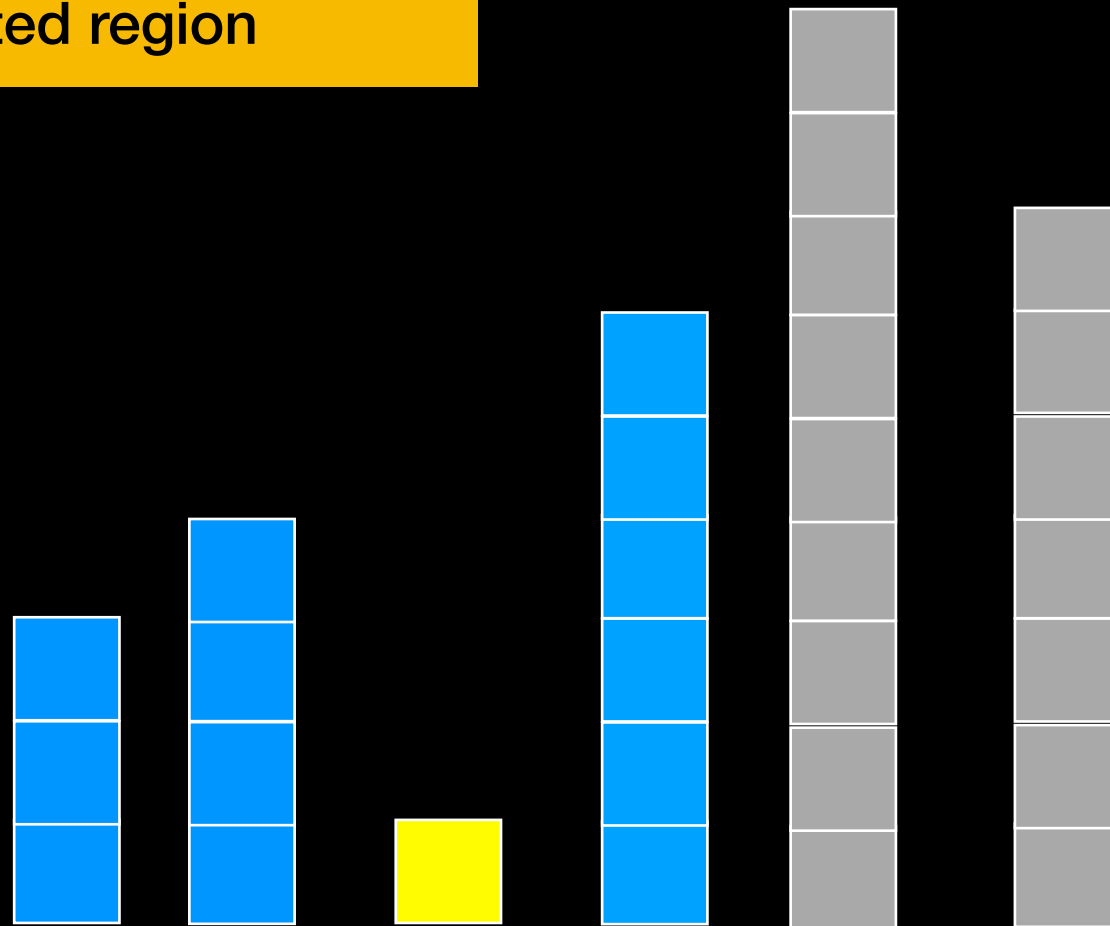
Insertion Sort



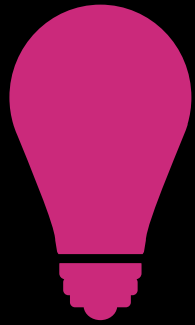
Pick first element in unsorted region and put it in right place in sorted region



3rd Pass



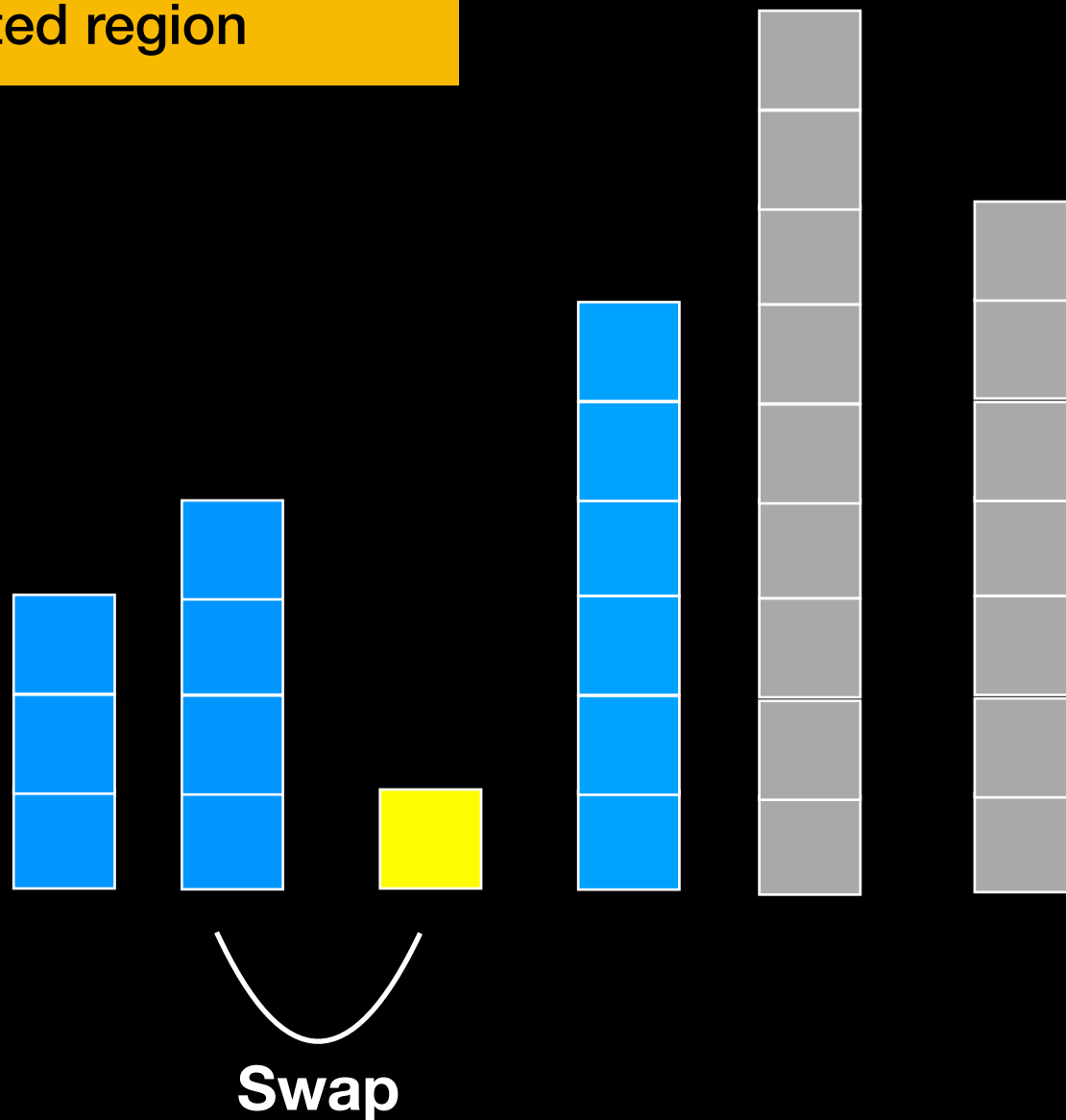
Insertion Sort



Pick first element in unsorted region and put it in right place in sorted region



3rd Pass



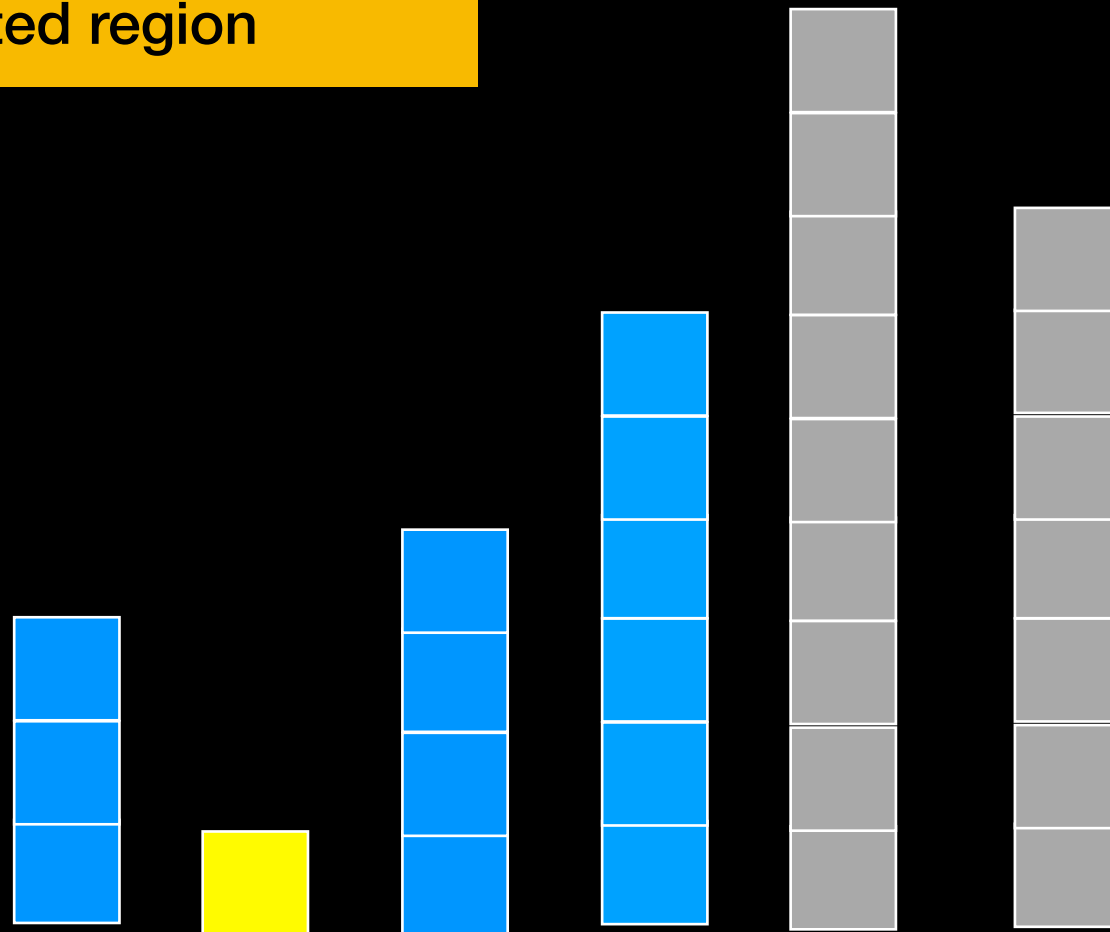
Insertion Sort



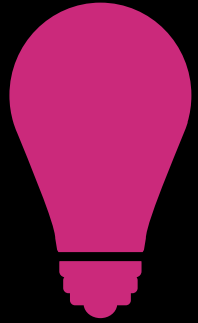
Pick first element in unsorted region and put it in right place in sorted region

Unsorted
Sorted

3rd Pass



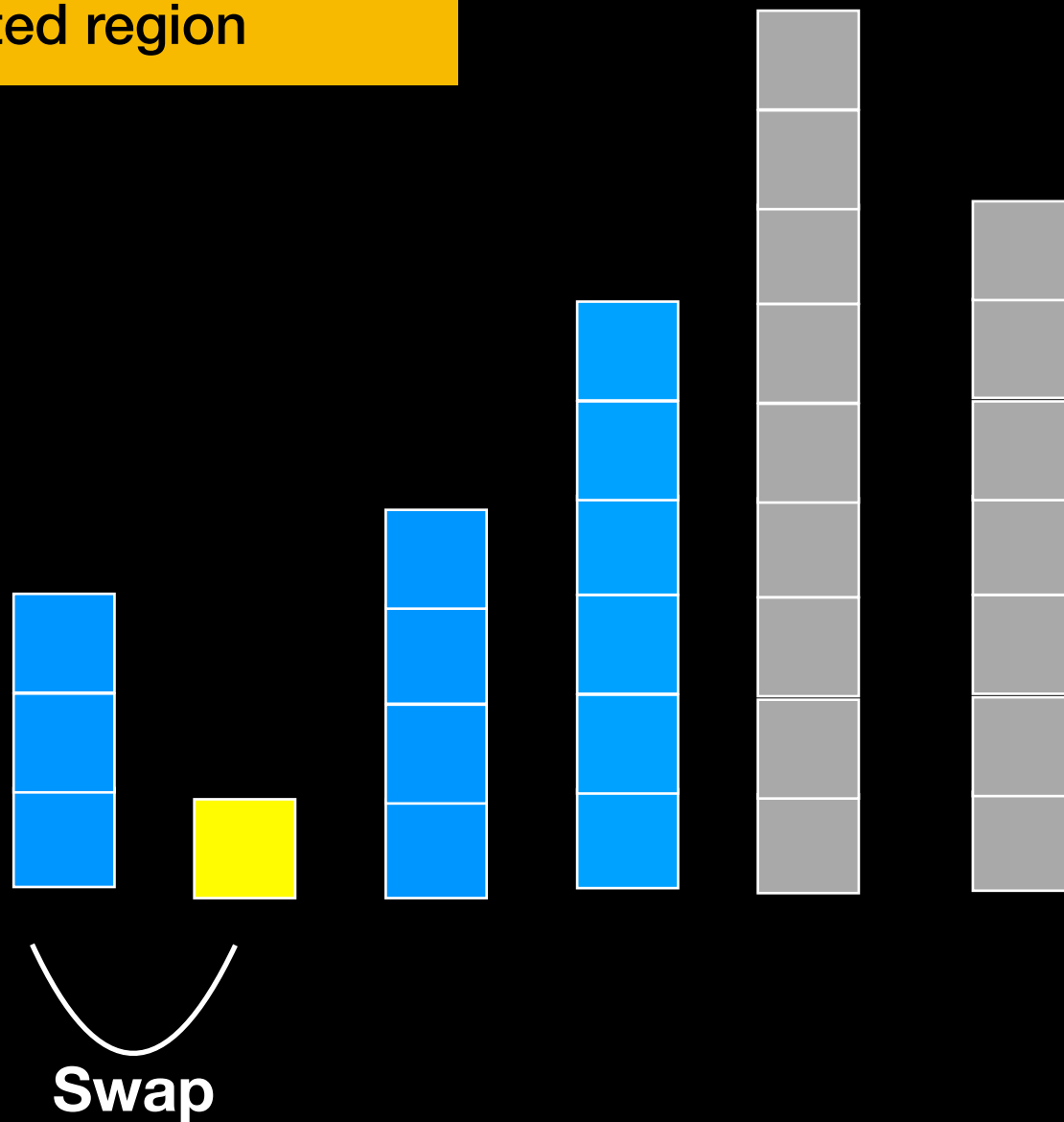
Insertion Sort



Pick first element in unsorted region and put it in right place in sorted region



3rd Pass



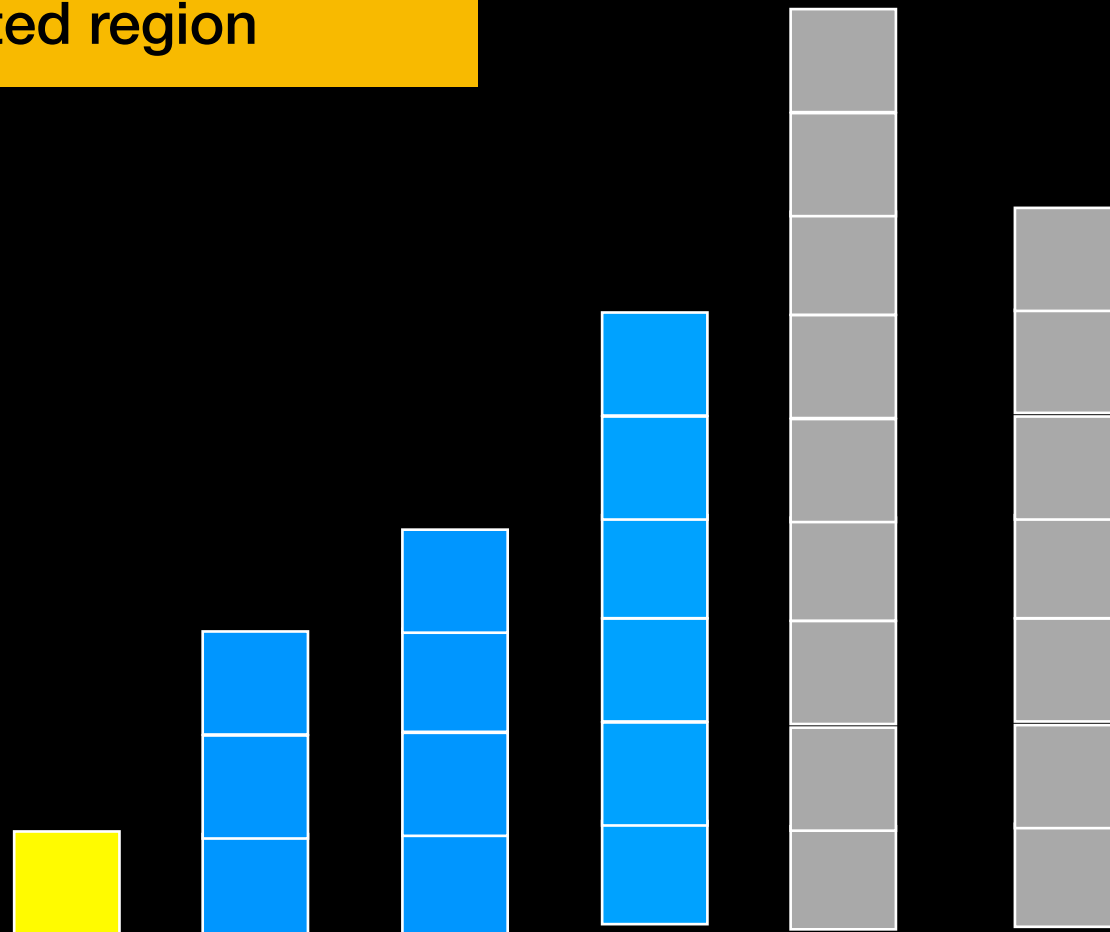
Insertion Sort



Pick first element in unsorted region and put it in right place in sorted region

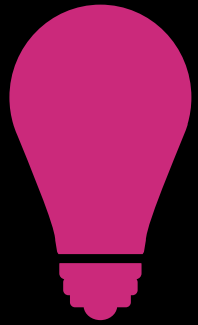


3rd Pass

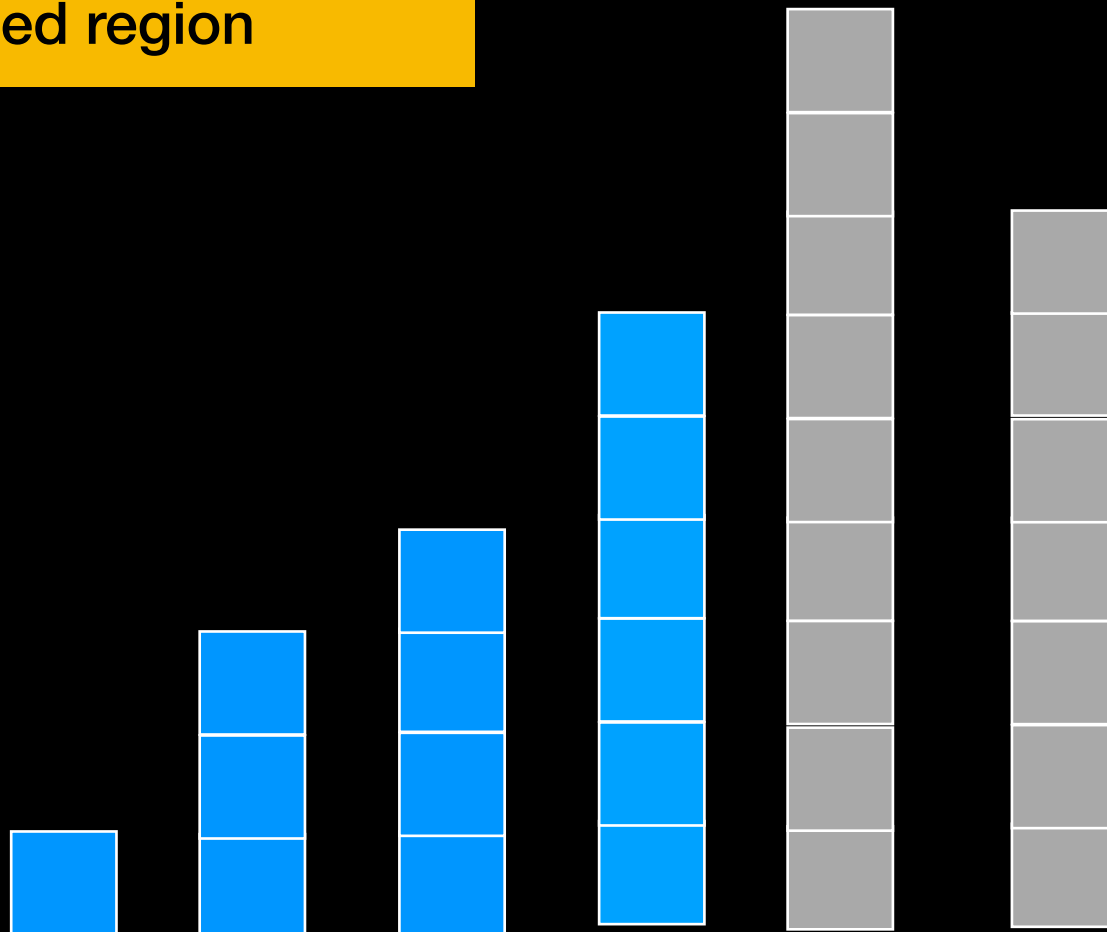


Insertion Sort

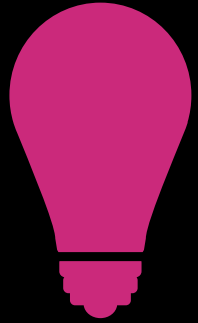
 Unsorted
 Sorted



Pick first element in unsorted region and put it in right place in sorted region



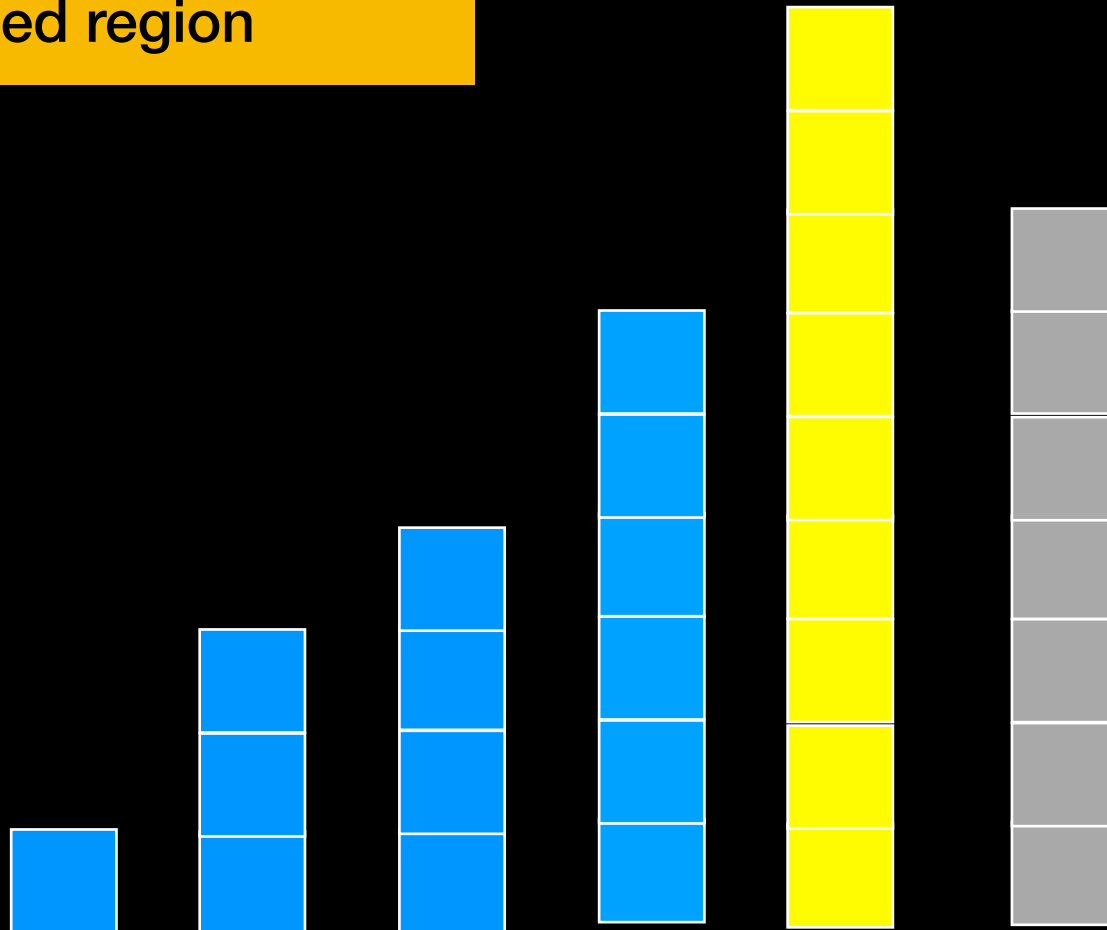
Insertion Sort



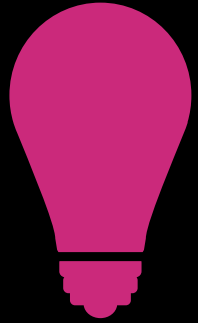
Pick first element in unsorted region and put it in right place in sorted region



4th Pass



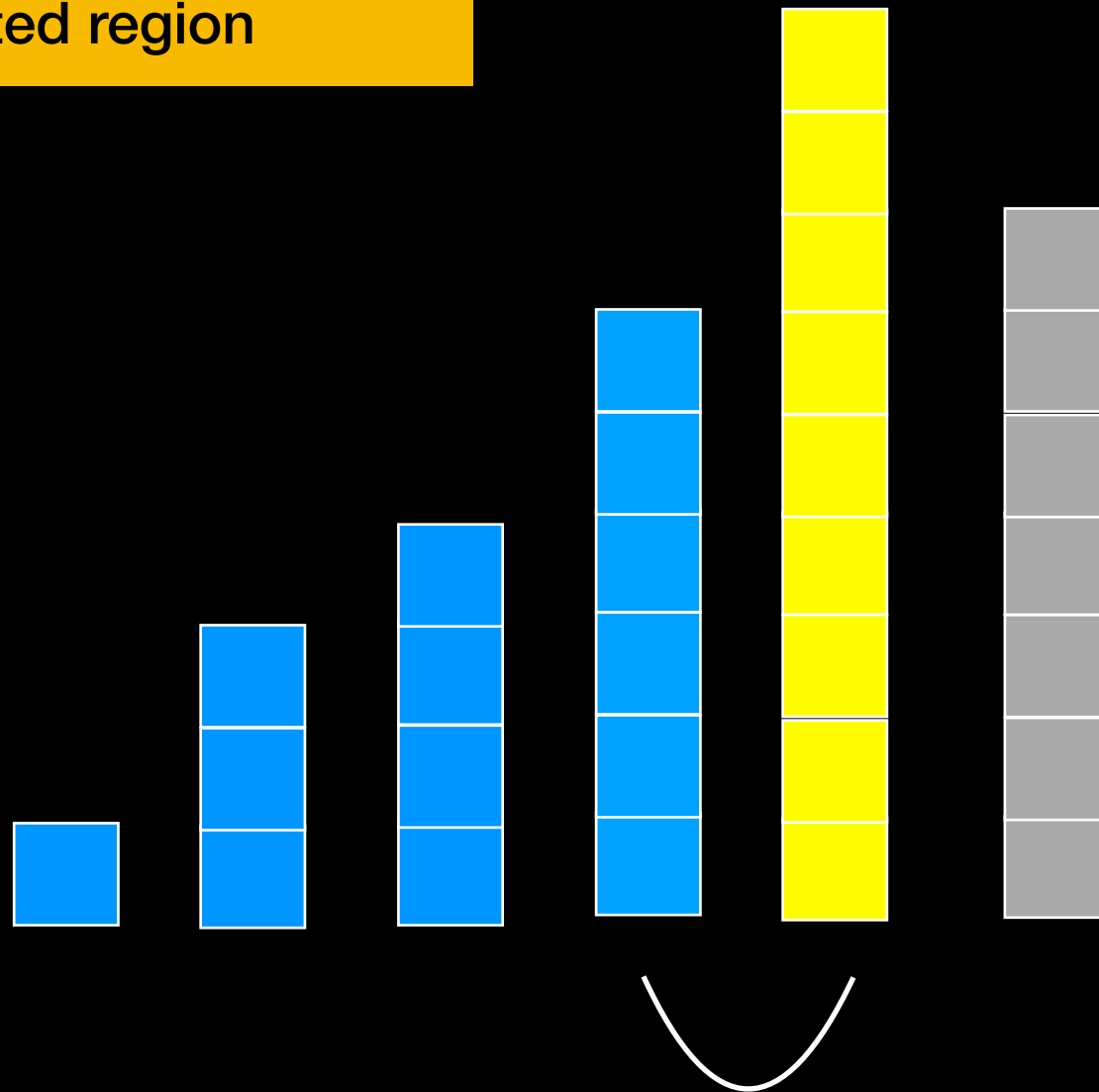
Insertion Sort




Pick first element in unsorted region and put it in right place in sorted region

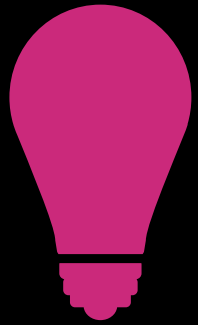


4th Pass

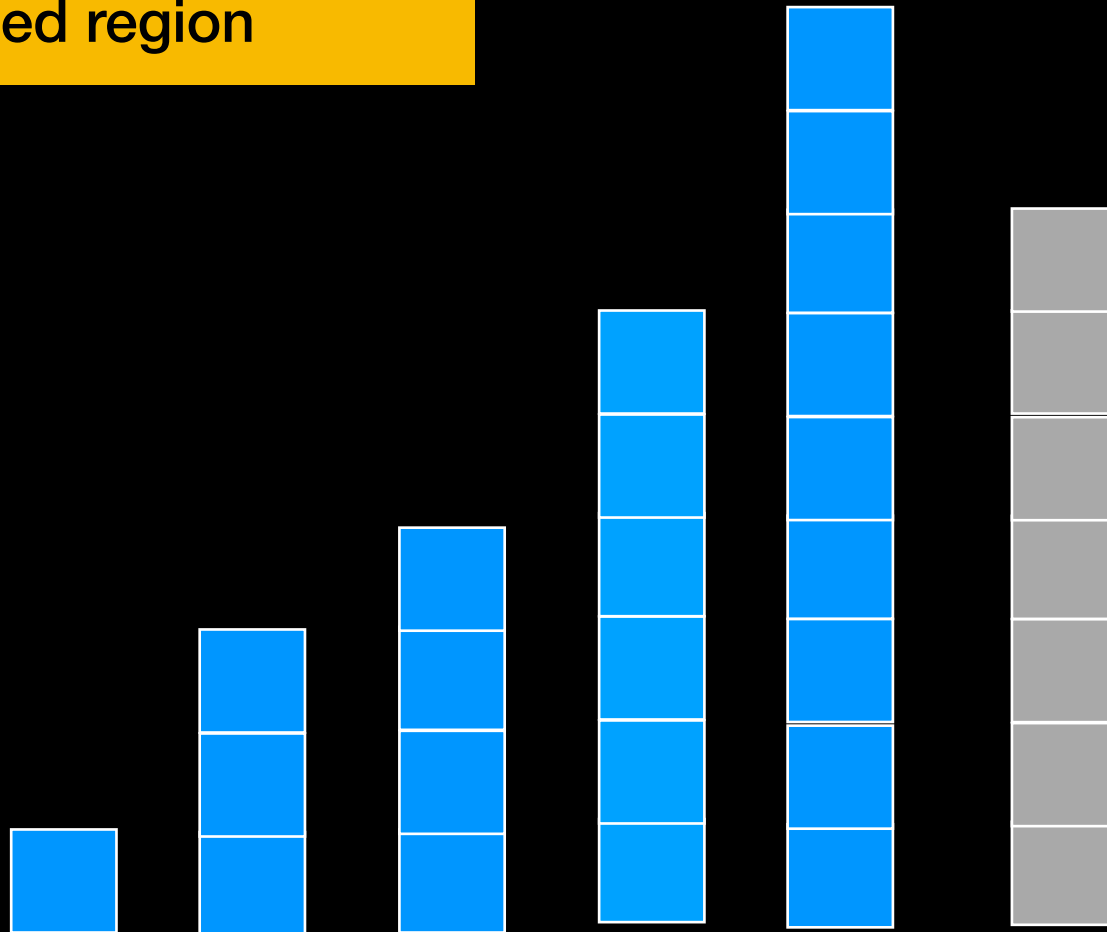


Insertion Sort

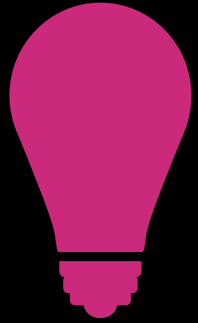
 Unsorted
 Sorted



Pick first element in unsorted region and put it in right place in sorted region



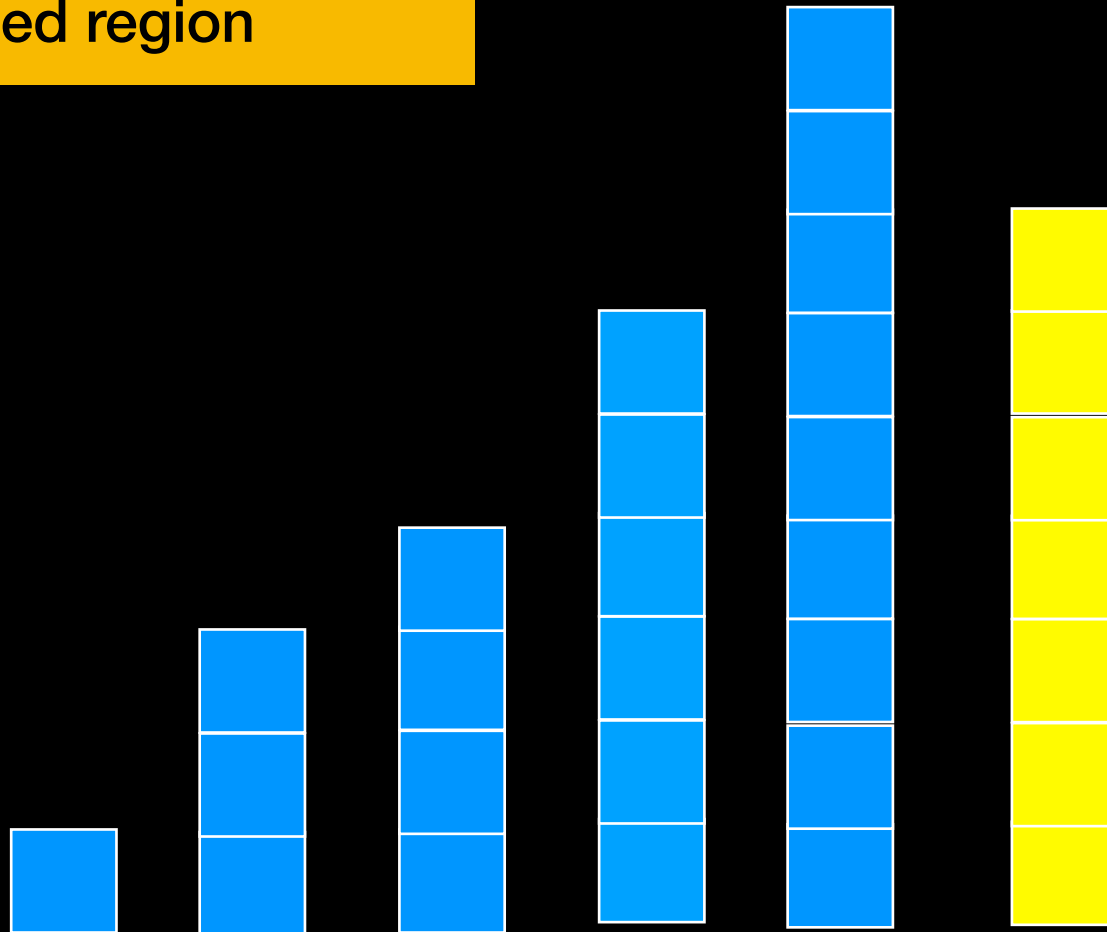
Insertion Sort



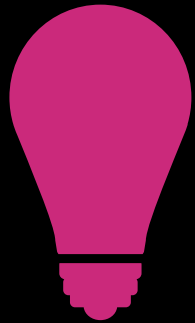
Pick first element in unsorted region and put it in right place in sorted region

Unsorted
Sorted

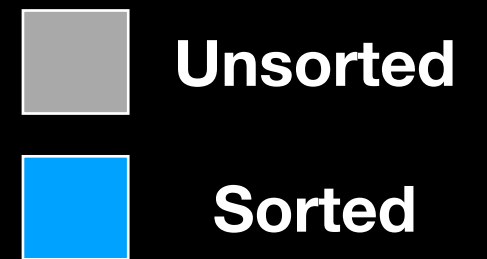
5th Pass



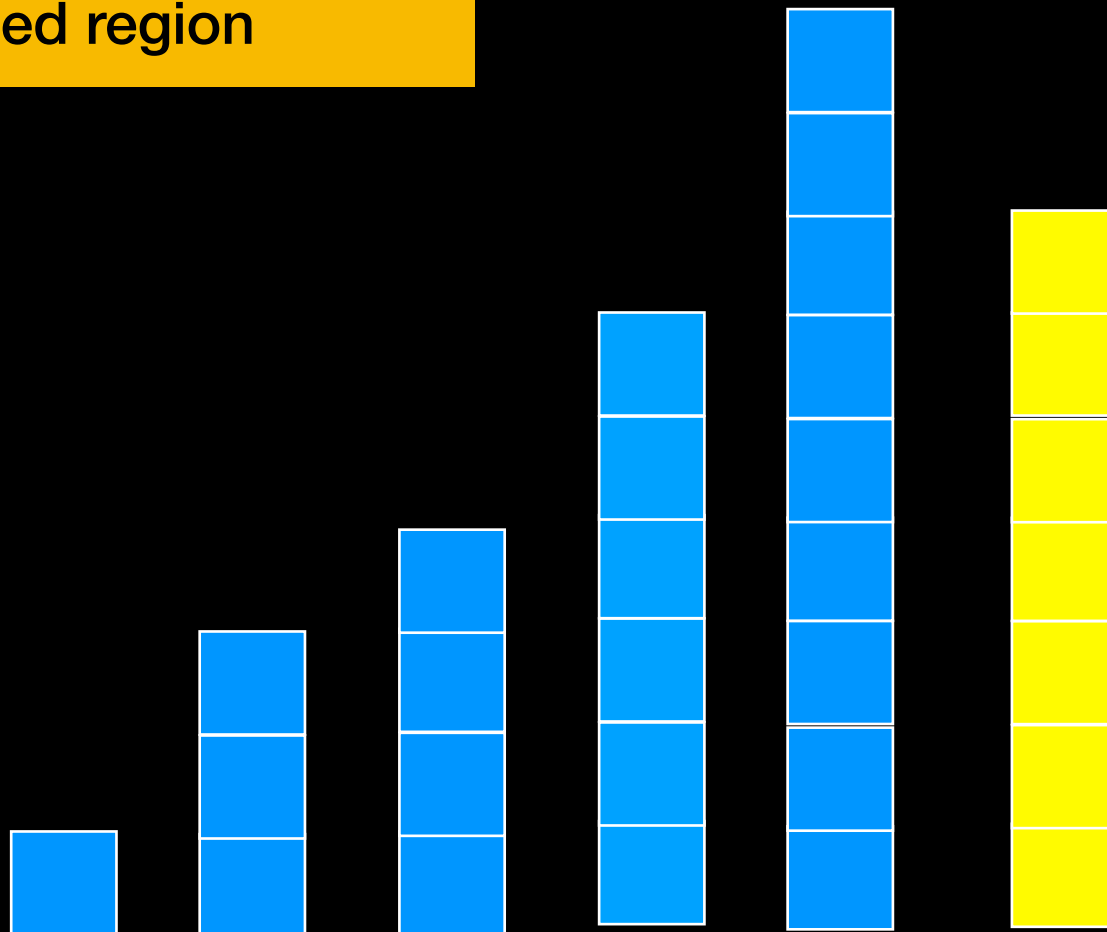
Insertion Sort



Pick first element in unsorted region and put it in right place in sorted region

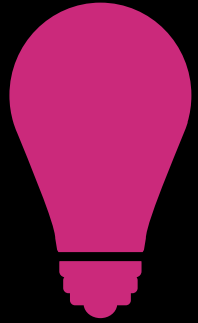


5th Pass



Swap

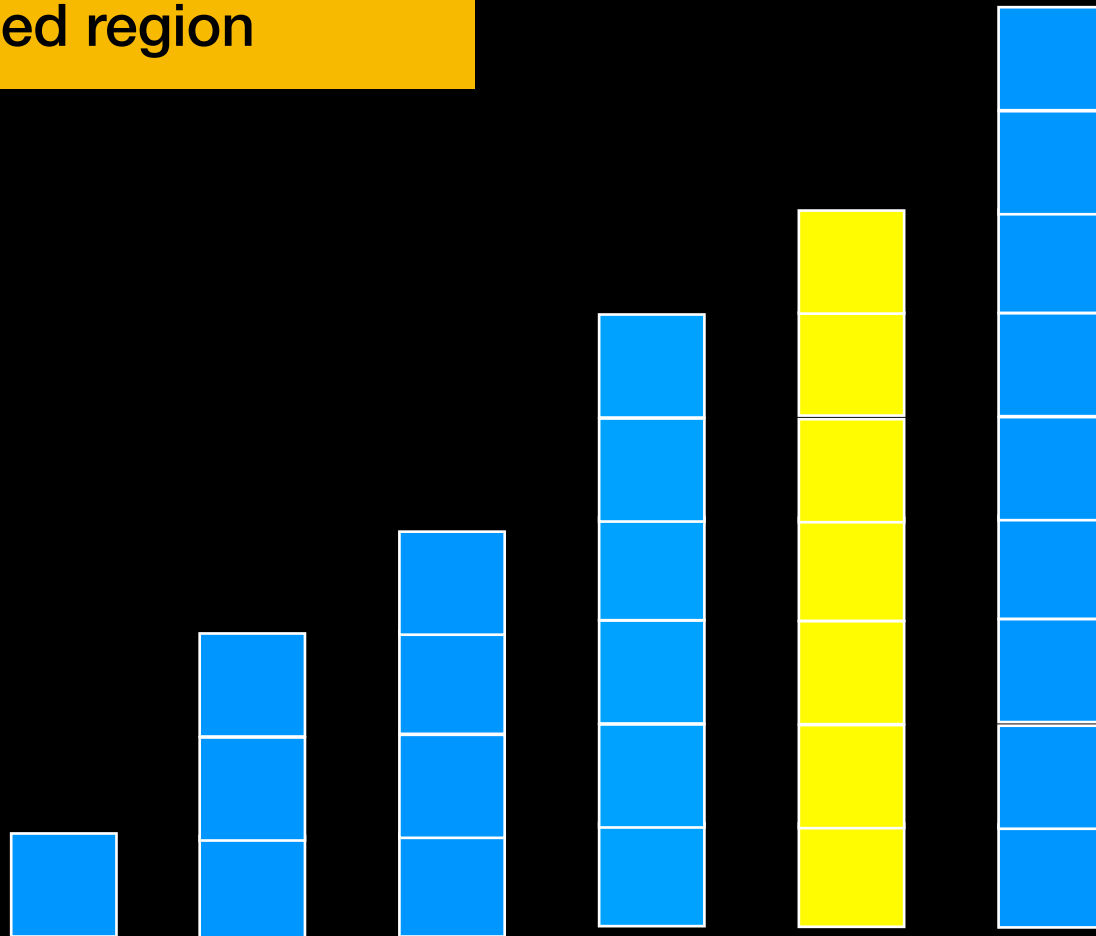
Insertion Sort



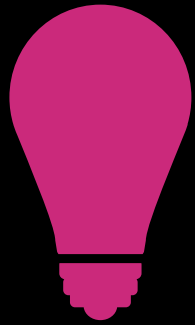
Pick first element in unsorted region and put it in right place in sorted region



5th Pass



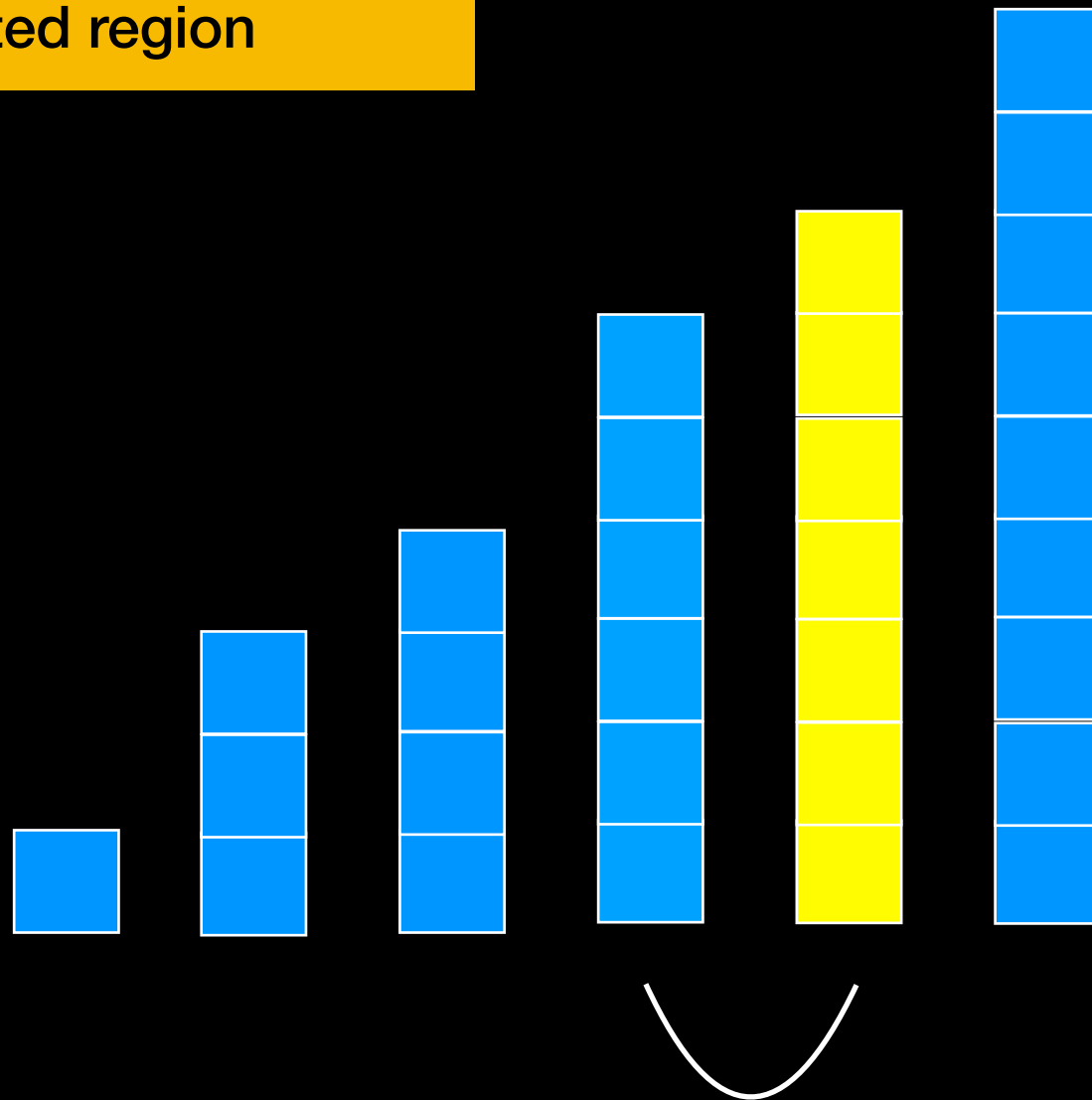
Insertion Sort



Pick first element in unsorted region and put it in right place in sorted region

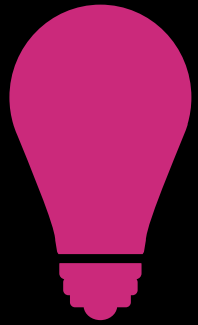
Unsorted
Sorted

5th Pass

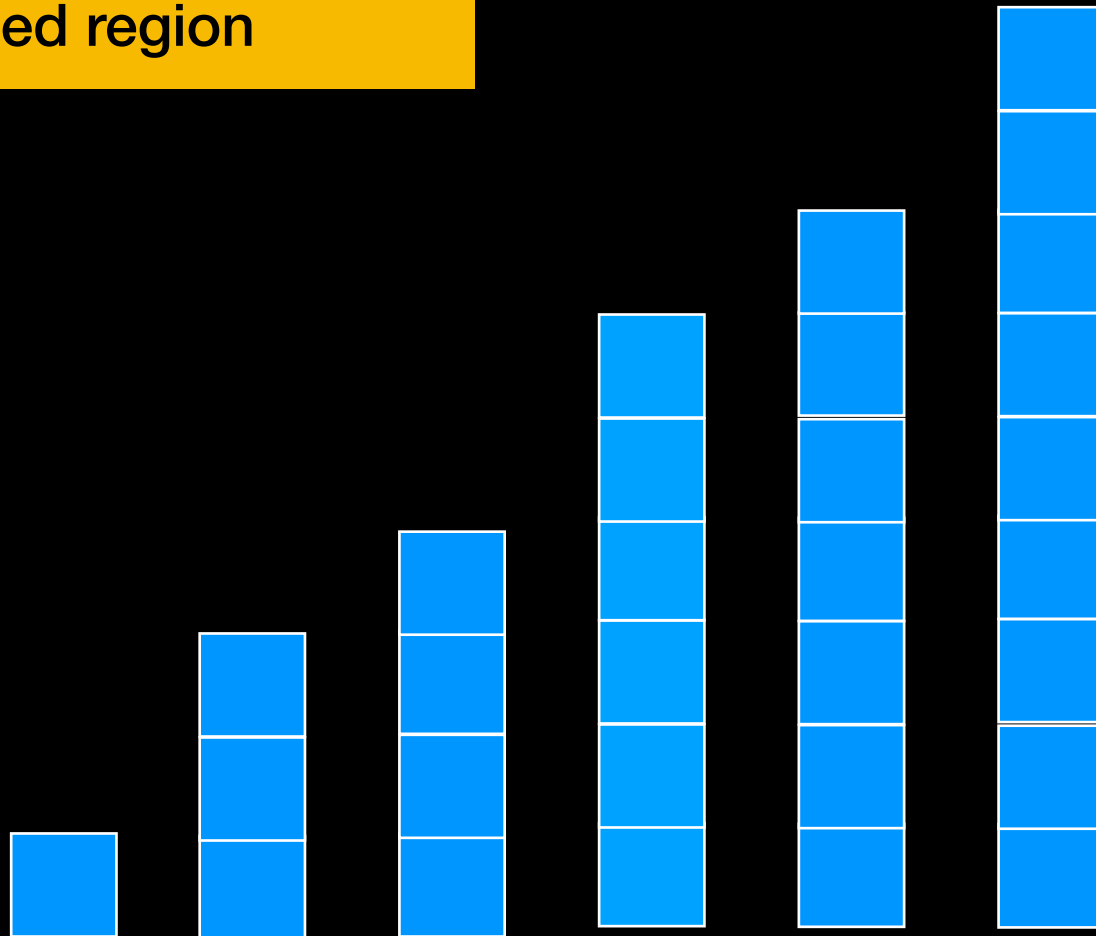


Insertion Sort

 Unsorted
 Sorted



Pick first element in unsorted region and put it in right place in sorted region



Insertion Sort Analysis

How much work?

First pass: **1** comparison and **at most 1** swap

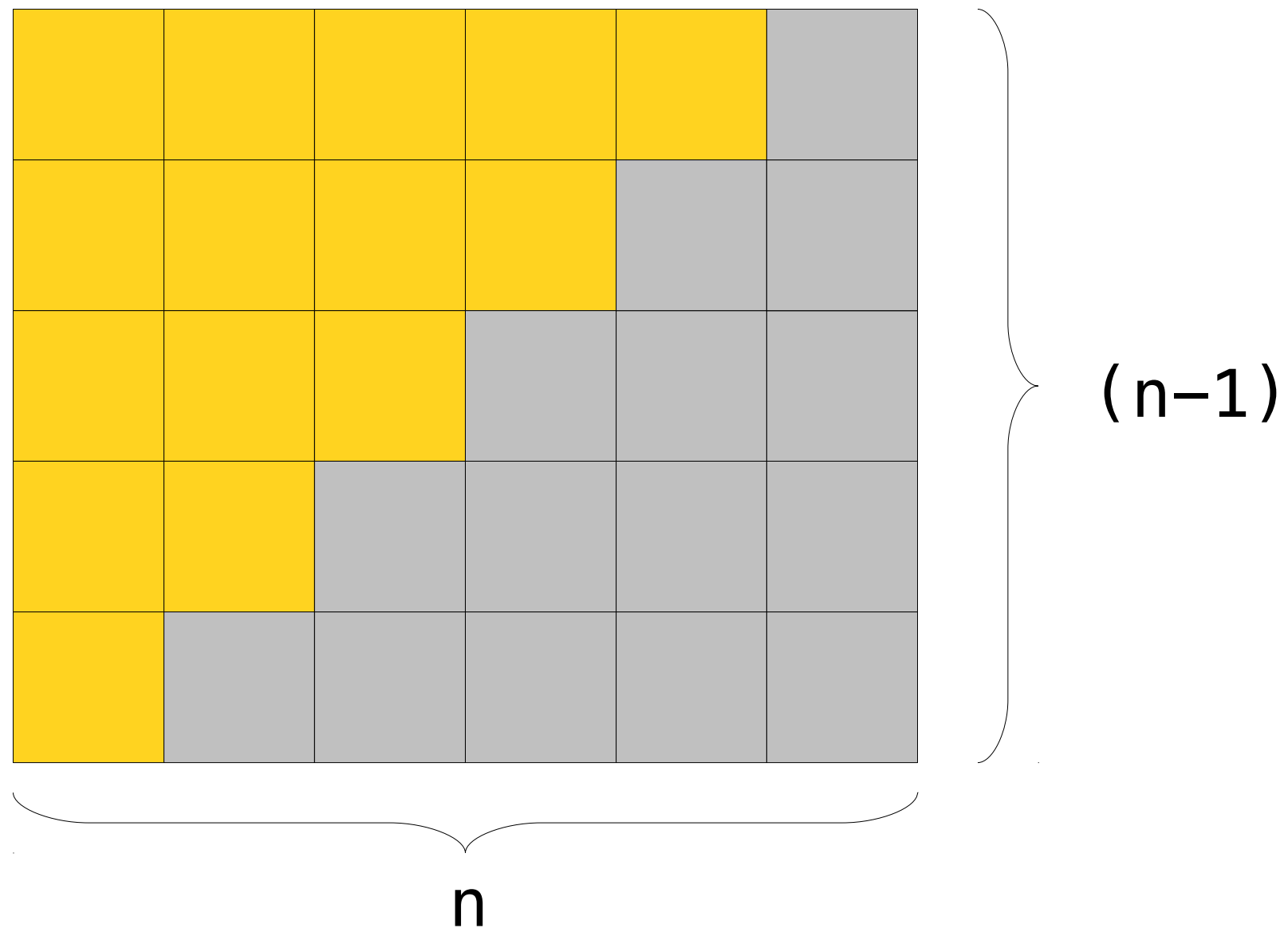
Second pass: **at most 2** comparisons and **at most 2** swaps

Third pass: **at most 3** comparisons and **at most 3** swaps

...

Total work: **$1 + 2 + 3 + \dots + (n-1)$**

$$1 + 2 + \dots + (n-2) + (n-1) = n(n-1)/2$$



Insertion Sort Analysis


$$T(n) = n(n-1) / 2 \text{ comparisons} + n(n-1) / 2 \text{ swaps} = O(\text{ })?$$

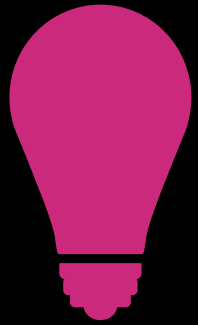
$$T(n) = 2((n^2-n) / 2) = O(\text{ })?$$

$$T(n) = n^2 - n = O(n^2)$$

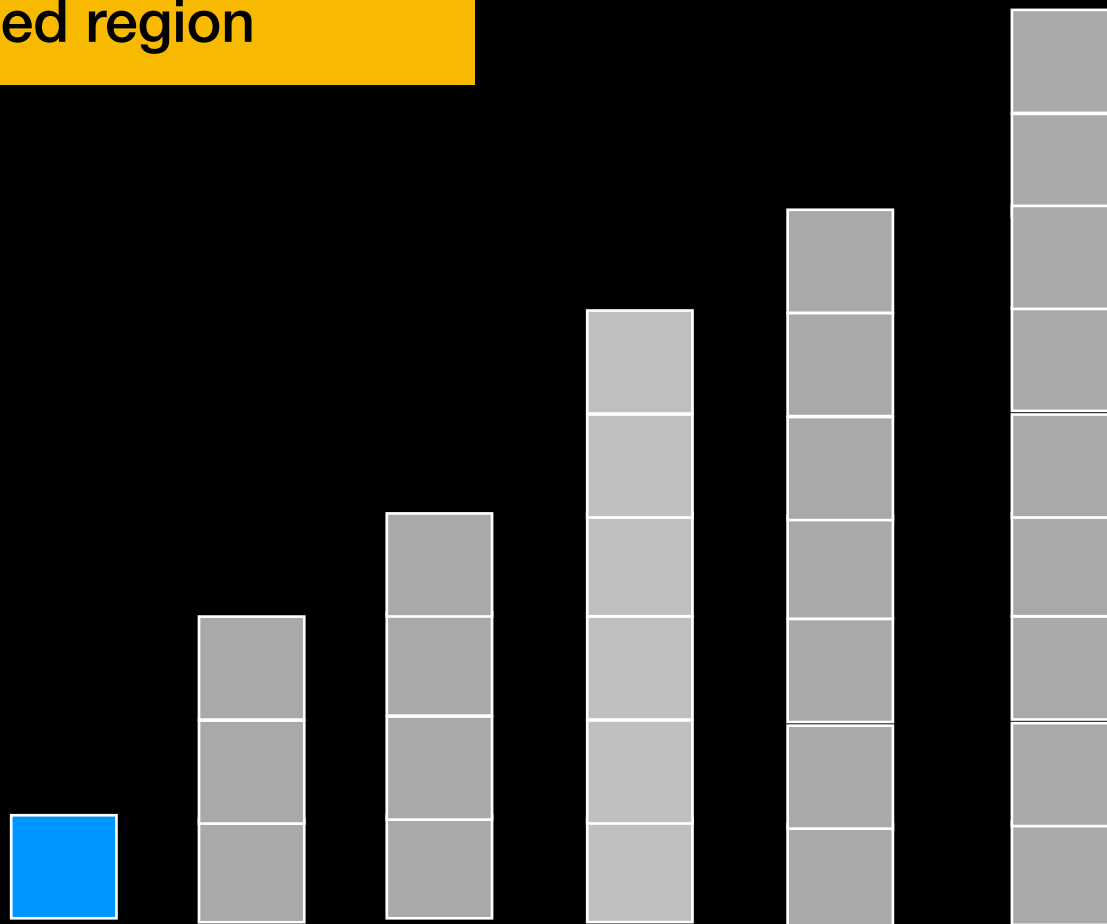
Insertion Sort run time is $O(n^2)$

Insertion Sort


 Unsorted
 Sorted

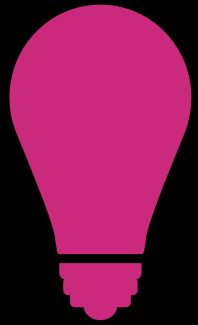


Pick first element in unsorted region and put it in right place in sorted region

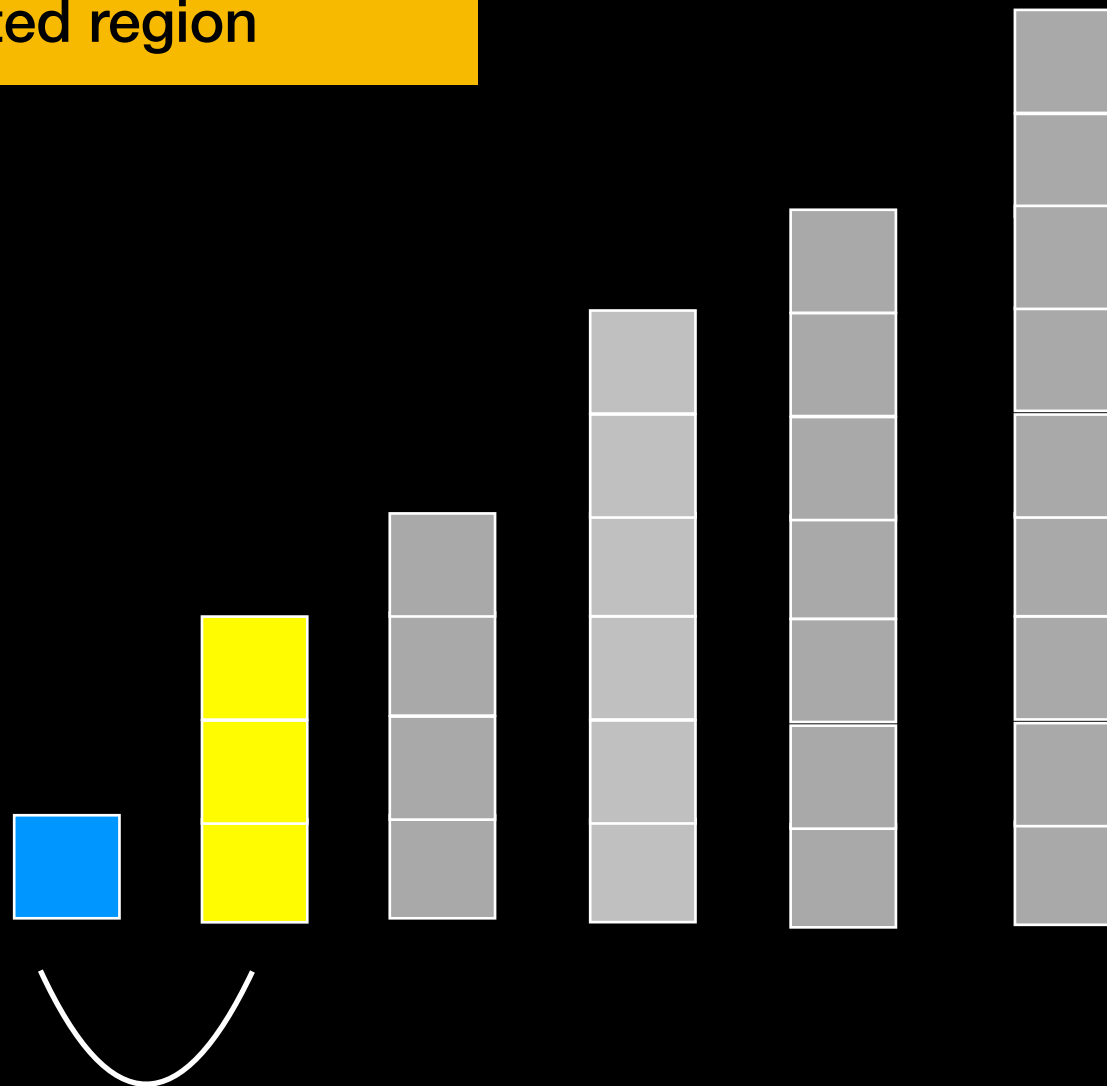


Insertion Sort



 Unsorted
 Sorted

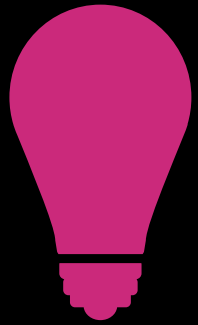


Pick first element in unsorted region and put it in right place in sorted region

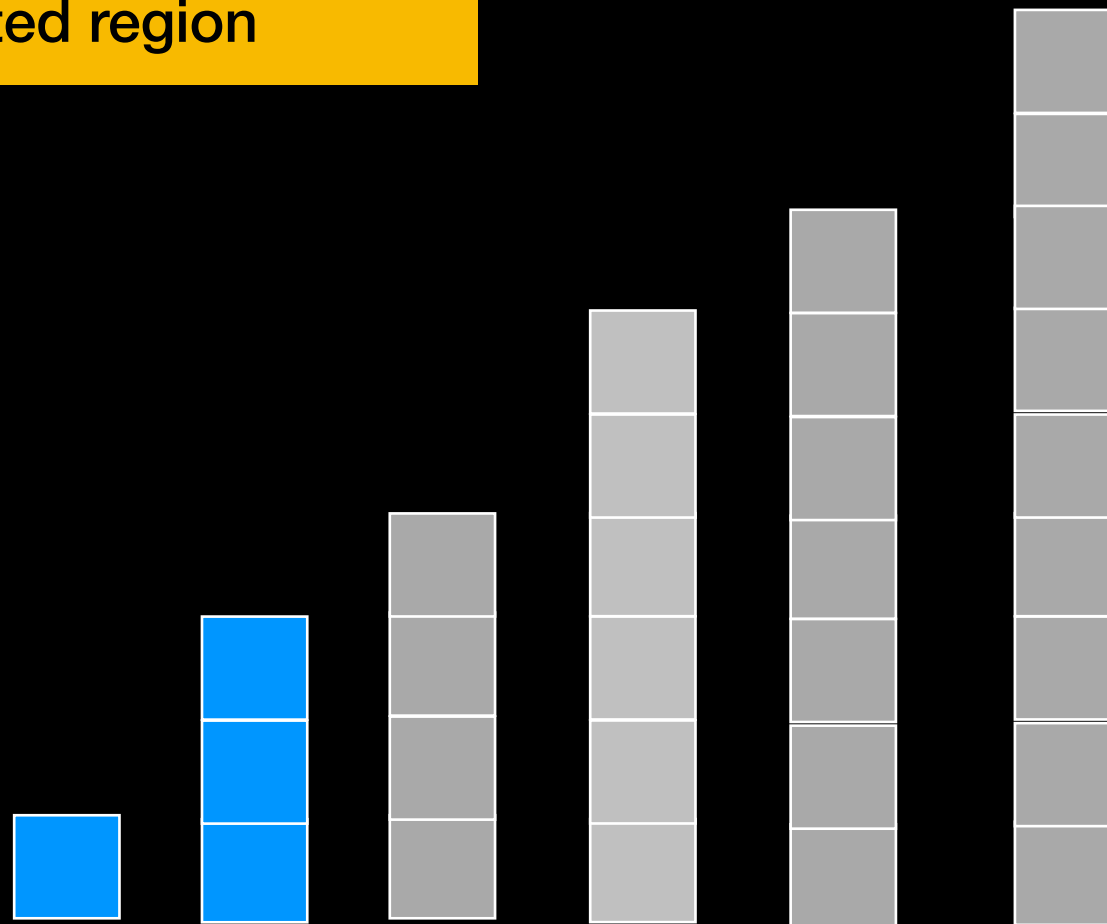


Insertion Sort

 Unsorted
 Sorted

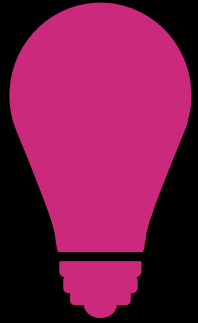


Pick first element in unsorted region and put it in right place in sorted region

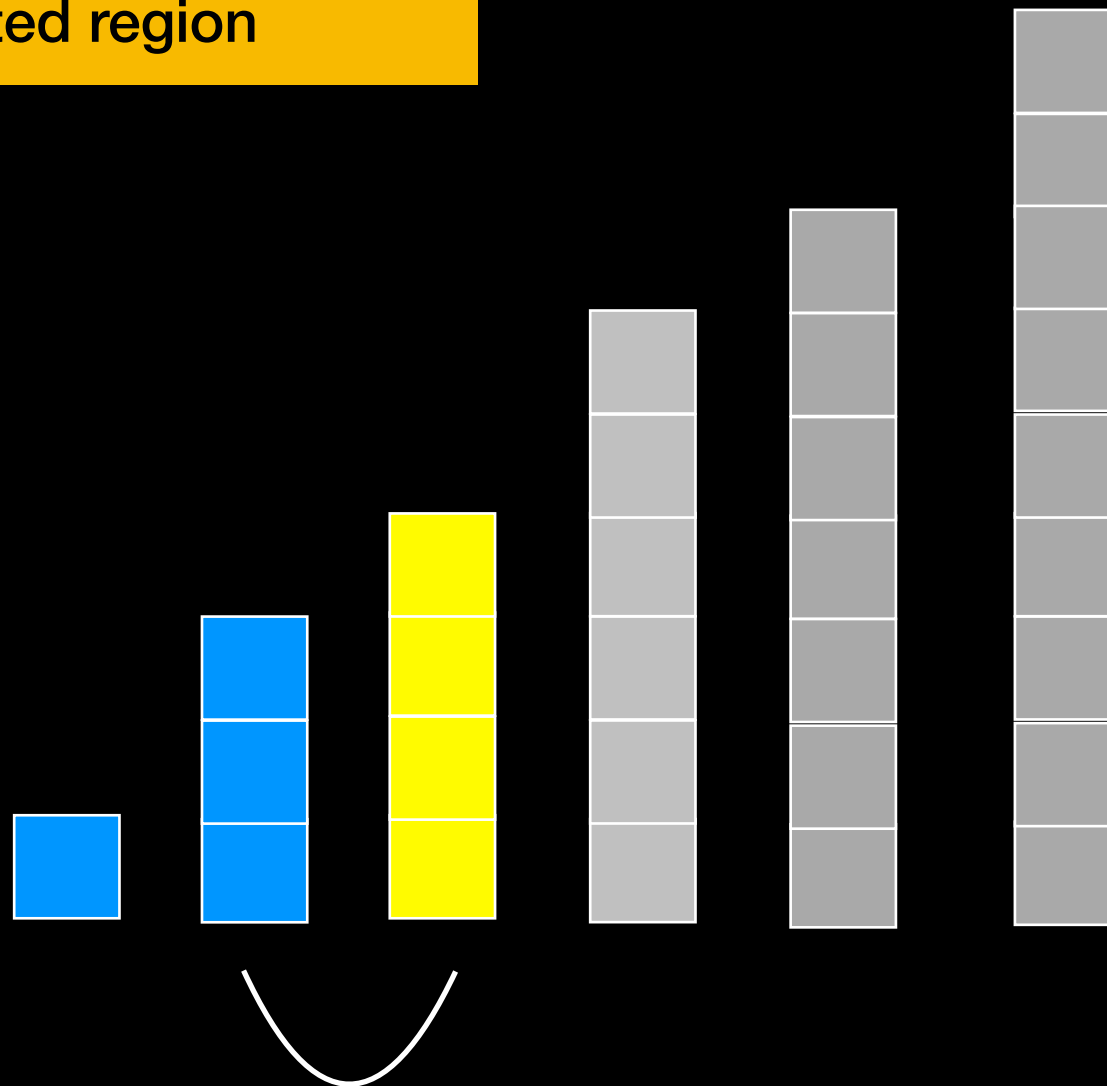


Insertion Sort

 Unsorted
 Sorted

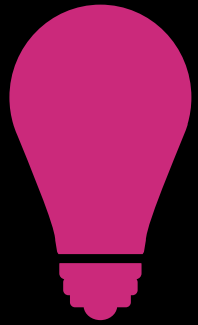


Pick first element in unsorted region and put it in right place in sorted region

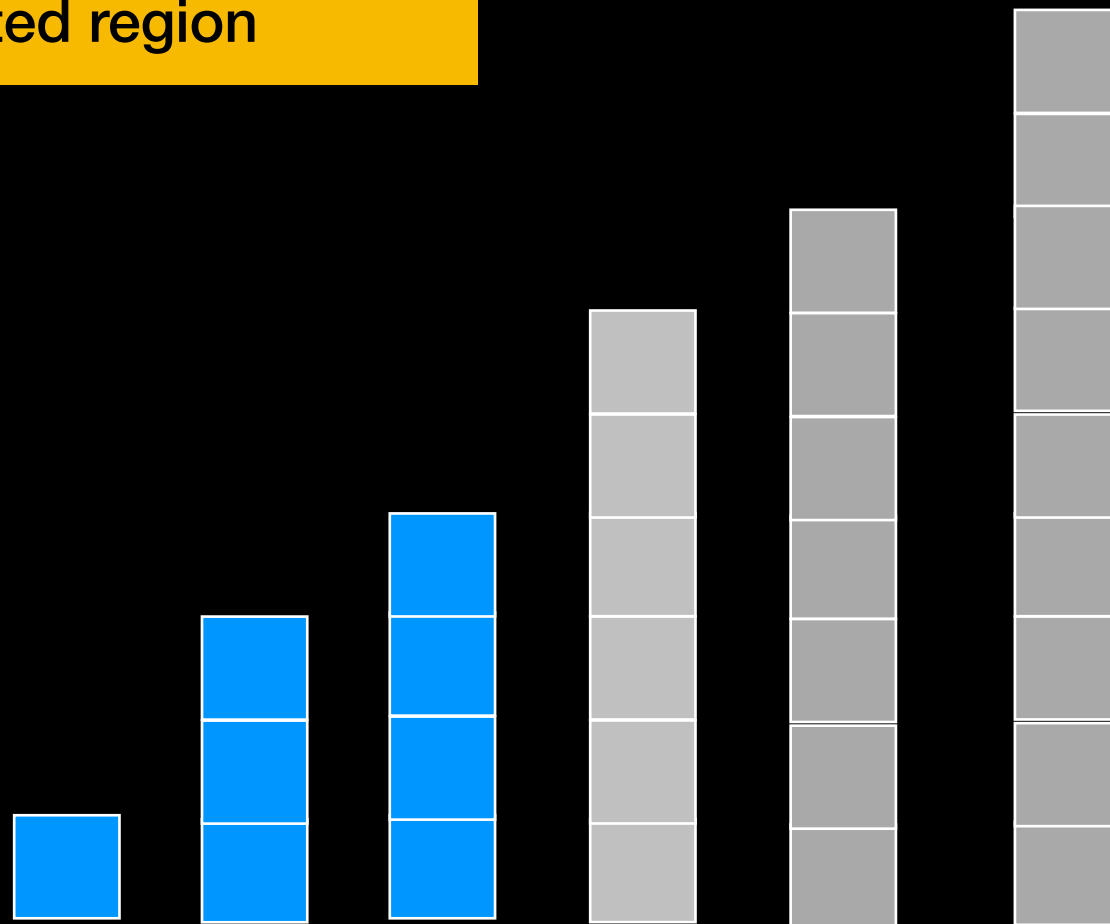


Insertion Sort

 Unsorted
 Sorted

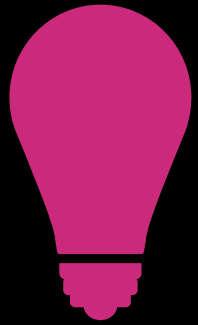


Pick first element in unsorted region and put it in right place in sorted region

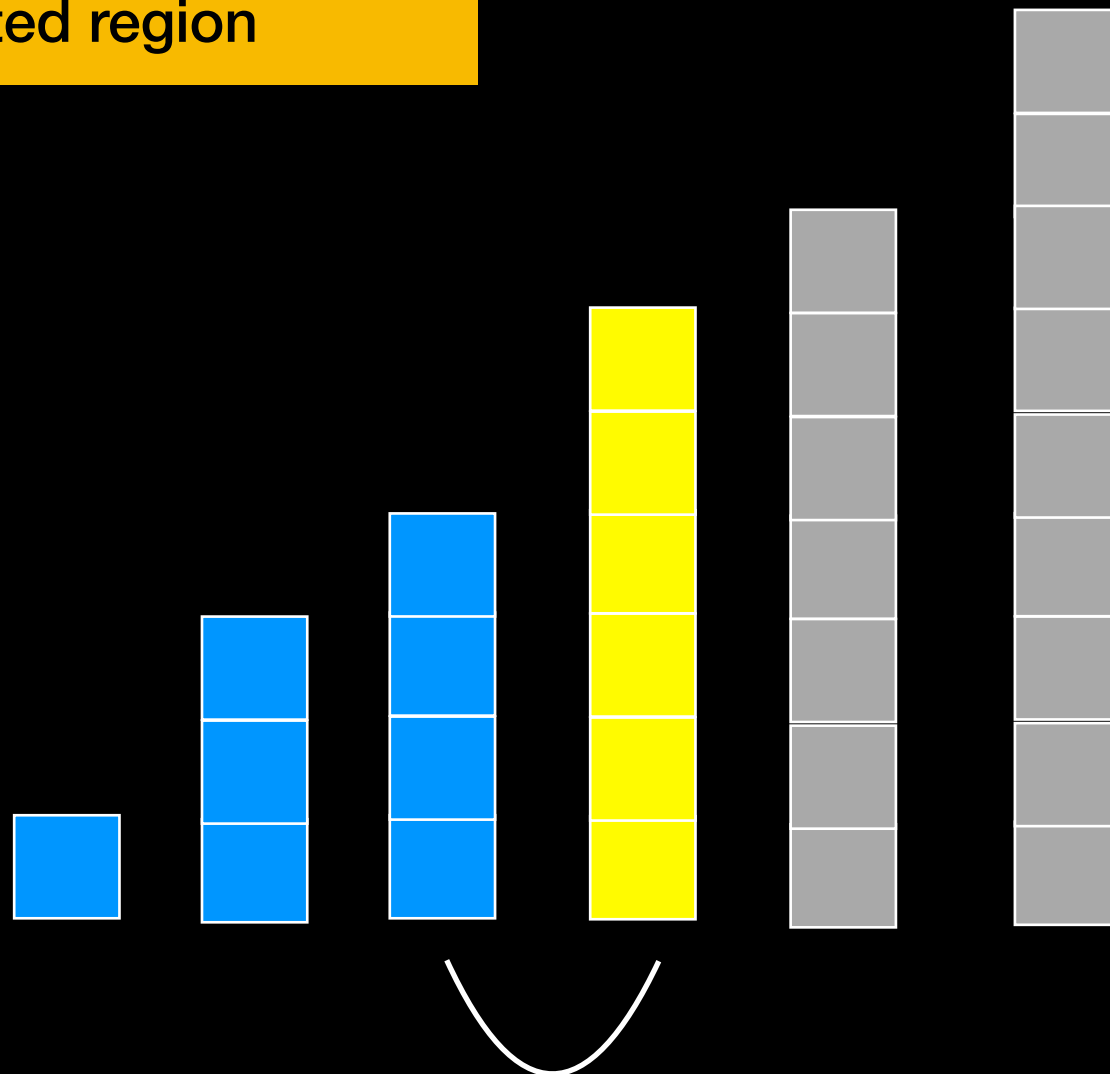


Insertion Sort

 Unsorted
 Sorted



Pick first element in unsorted region and put it in right place in sorted region

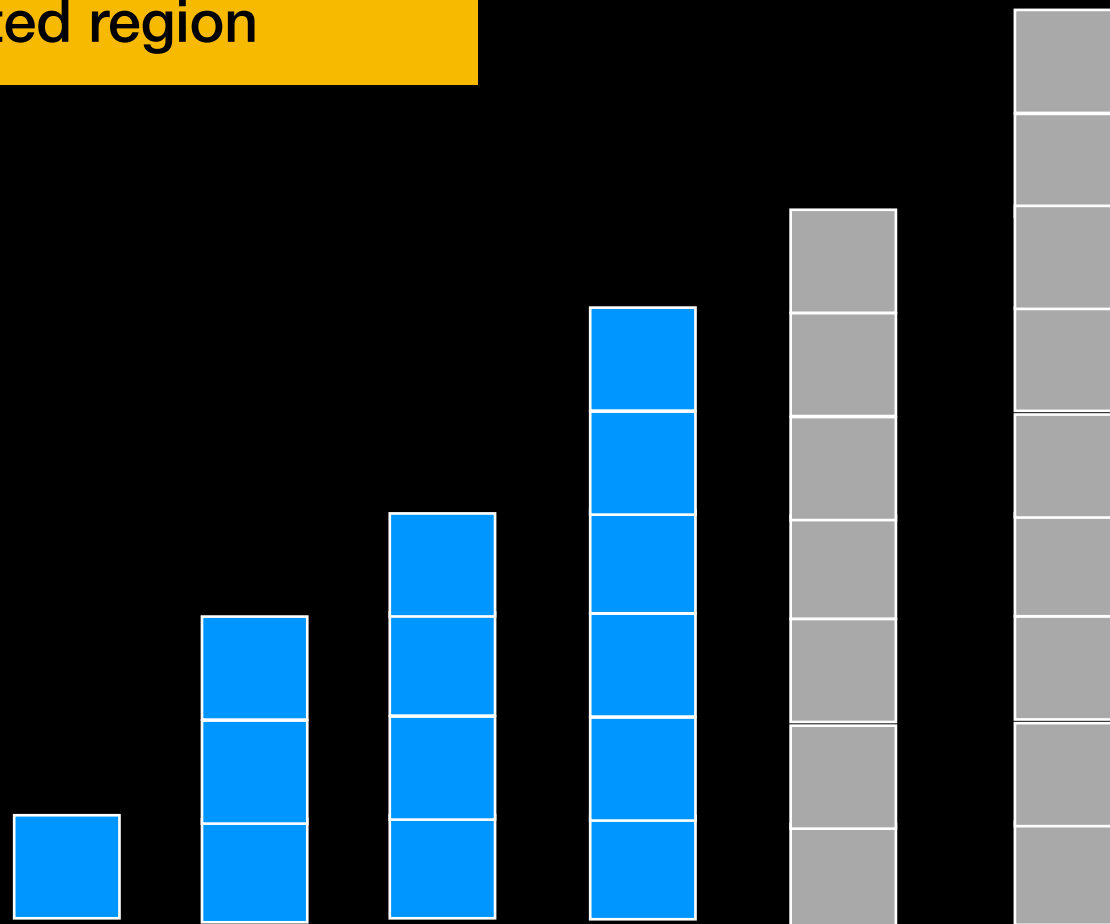


Insertion Sort



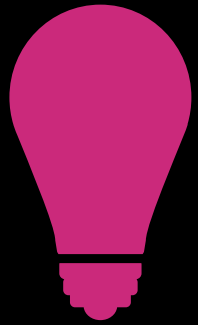
Pick first element in unsorted region and put it in right place in sorted region

Unsorted
Sorted

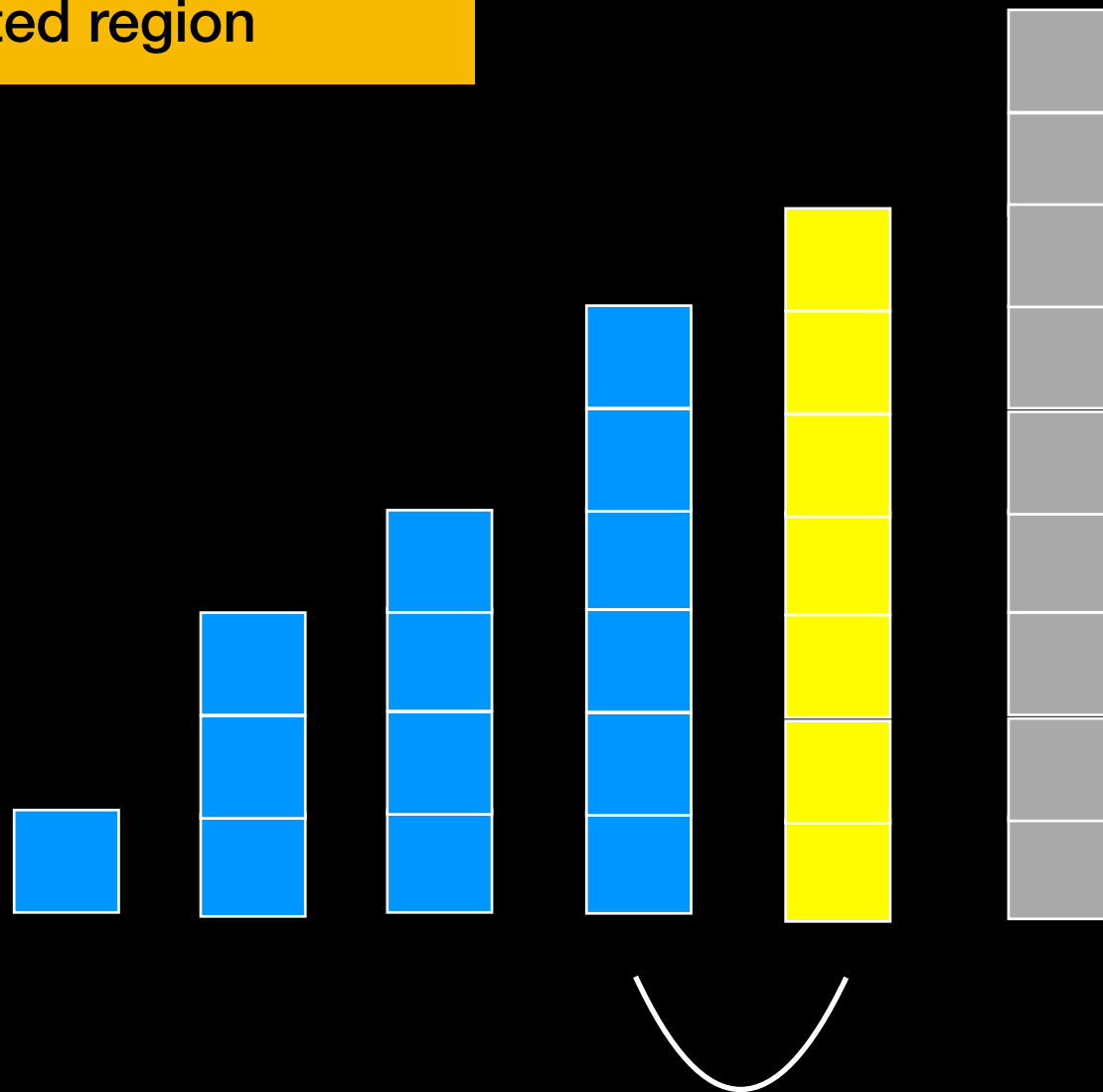


Insertion Sort

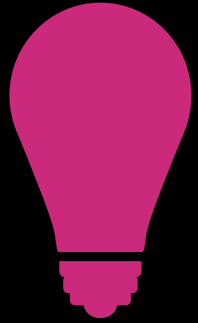
■ Unsorted
■ Sorted



Pick first element in unsorted region and put it in right place in sorted region

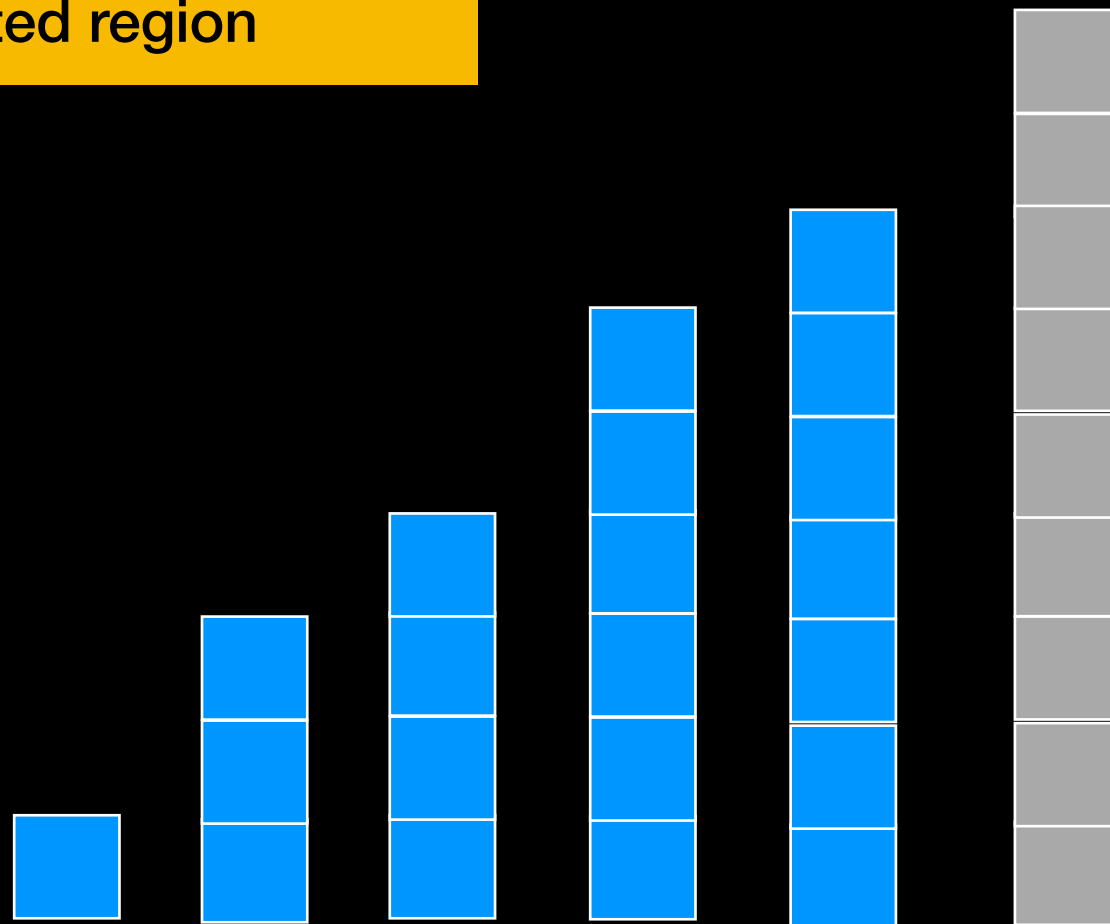


Insertion Sort




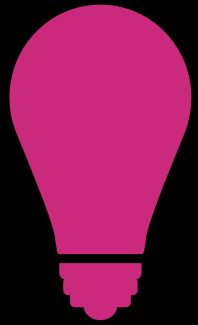
Pick first element in unsorted region and put it in right place in sorted region

Unsorted
Sorted

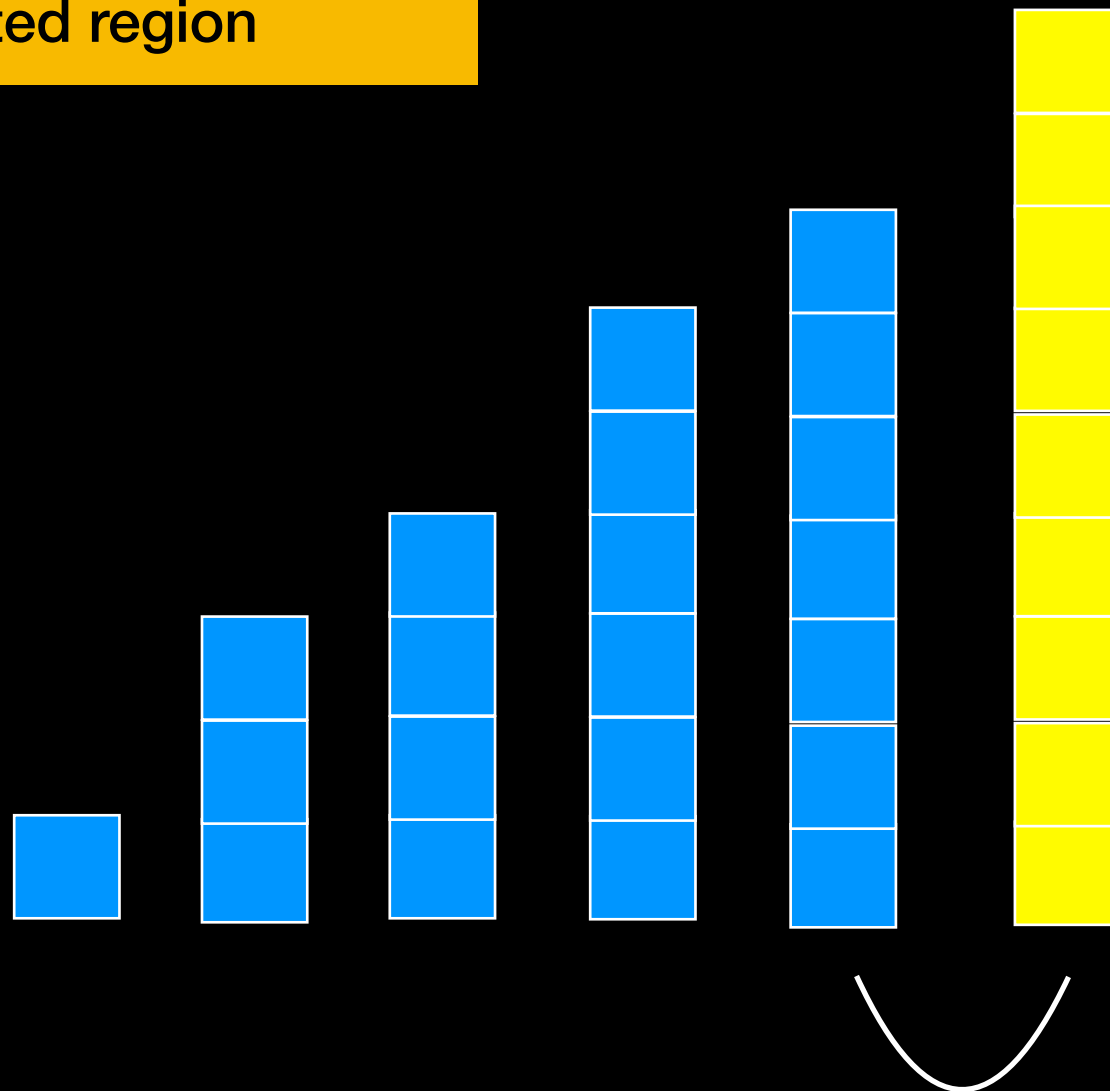


Insertion Sort



 Unsorted
 Sorted

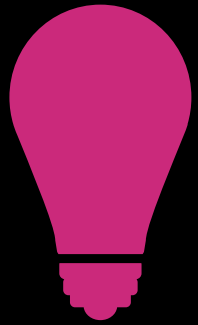


Pick first element in unsorted region and put it in right place in sorted region

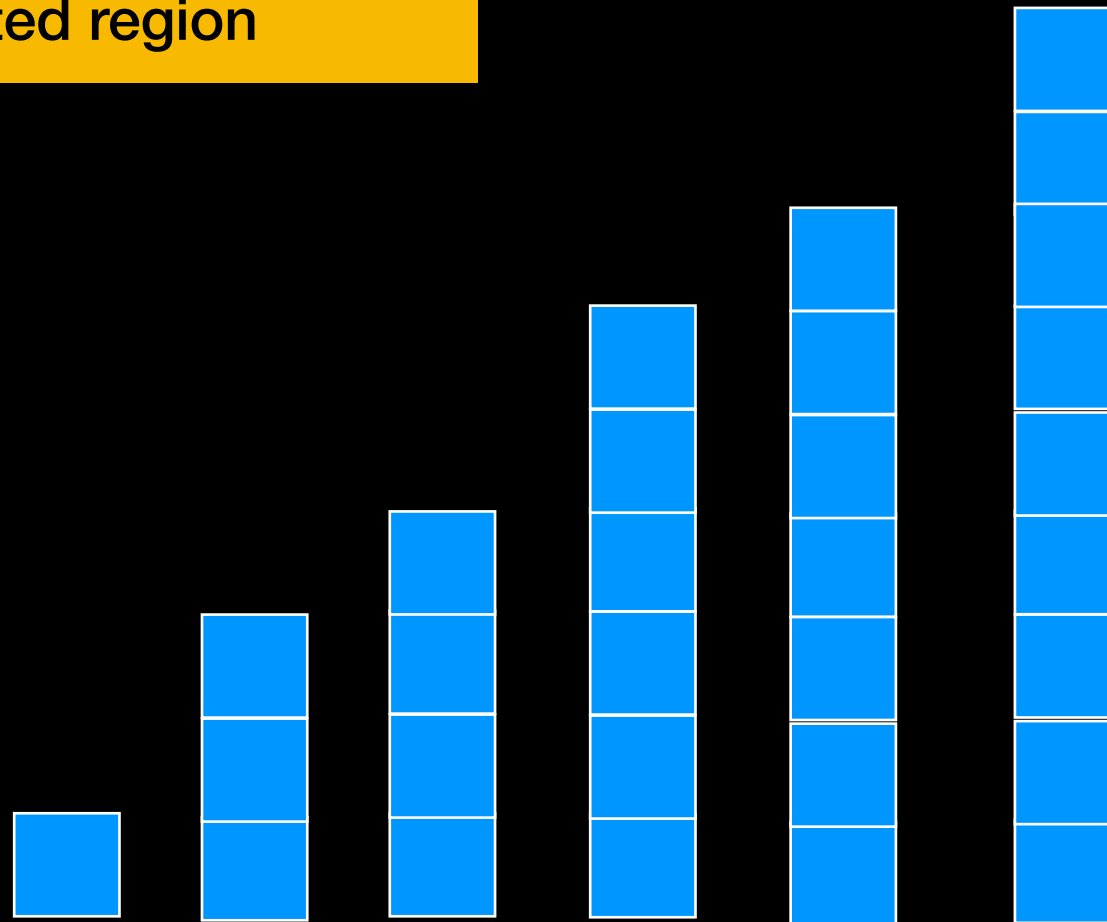


Insertion Sort

 Unsorted
 Sorted



Pick first element in unsorted region and put it in right place in sorted region



Insertion Sort Analysis

Execution time DOES depend on initial arrangement of data

Worst case: $O(n^2)$ comparisons and data moves

Best case: $O(n)$ comparisons and data moves

Stable

If array is already sorted Insertion sort will do only n comparisons and no swaps \Rightarrow good choice for **small n** and data likely **somewhat sorted**

```

template<typename ItemType>
void insertionSort(ItemType the_array[], size_t size)
{
    // unsorted = first index of the unsorted region,
    // Initially, sorted region is the_array[0],
    // unsorted region is the_array[1 ... size-1].
    // In general, sorted region is the_array[0 ... unsorted-1],
    // unsorted region the_array[unsorted ... size-1]
    for (int unsorted = 1; unsorted < size; unsorted++)
    {
        // At this point, the_array[0 ... unsorted-1] is sorted.
        // Keep swapping item to be inserted currently at
        // the_array[unsorted] with items at lower indices
        // as long as its value is > the index of the item
        int current = unsorted; // currently being inserted
        while ((current > 0) &&
            (the_array[current - 1] > the_array[current]))
        {
            swap(the_array[current], the_array[current - 1]); // swap
            current--;
        } // end while
    } // end for
} // end insertionSort

```

```

template<typename ItemType>
void insertionSort(ItemType the_array[], size_t size)
{
    // unsorted = first index of the unsorted region,
    // Initially, sorted region is the_array[0],
    // unsorted region is the_array[1 ... size-1].
    // In general, sorted region is the_array[0 ... unsorted-1],
    // unsorted region the_array[unsorted ... size-1]
    Pass for (int unsorted = 1; unsorted < size; unsorted++)
    O(n) {
        // At this point, the_array[0 ... unsorted-1] is sorted.
        // Keep swapping item to be inserted currently at
        // the_array[unsorted] with items at lower indices
        // as long as its value is > the index of the item
        int current = unsorted; // currently being inserted
        O(n) while ((current > 0) &&
                    (the_array[current - 1] > the_array[current]))
        {
            swap(the_array[current], the_array[current - 1]); // swap
            current--;
        } // end while
    } // end for
} // end insertionSort

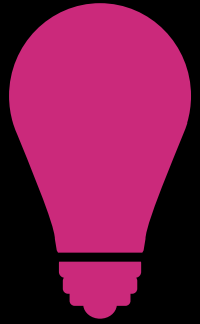
```

O(n²)

Raise your hand if you had
Insertion Sort

What we have so far

	Worst Case	Best Case
Selection Sort	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n)$

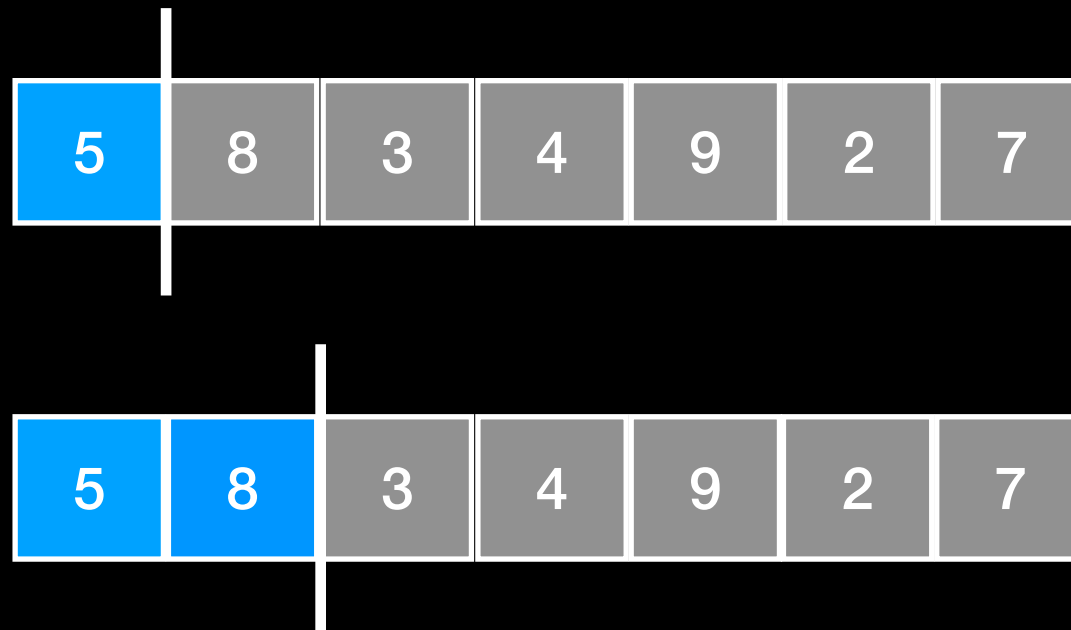


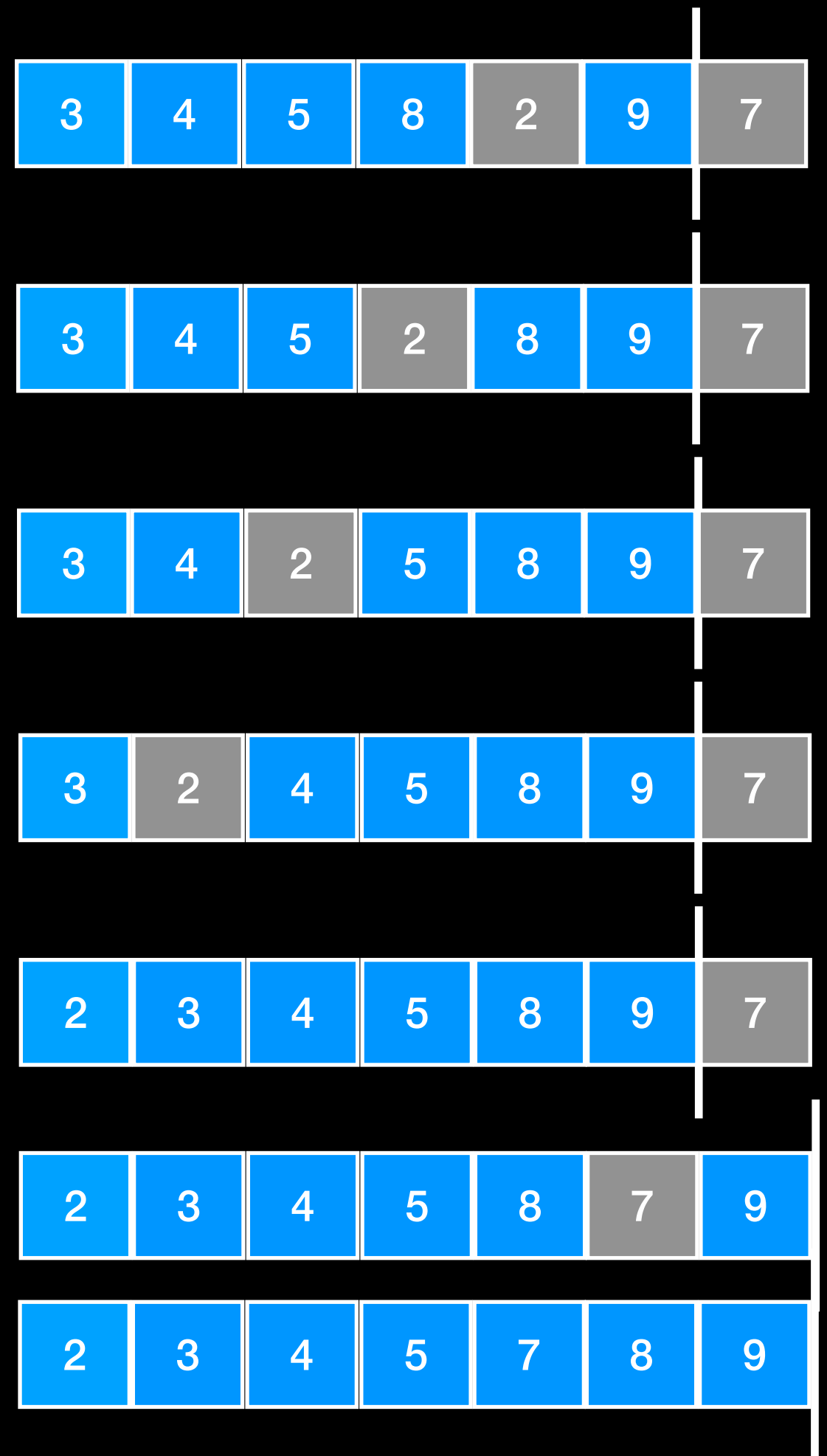
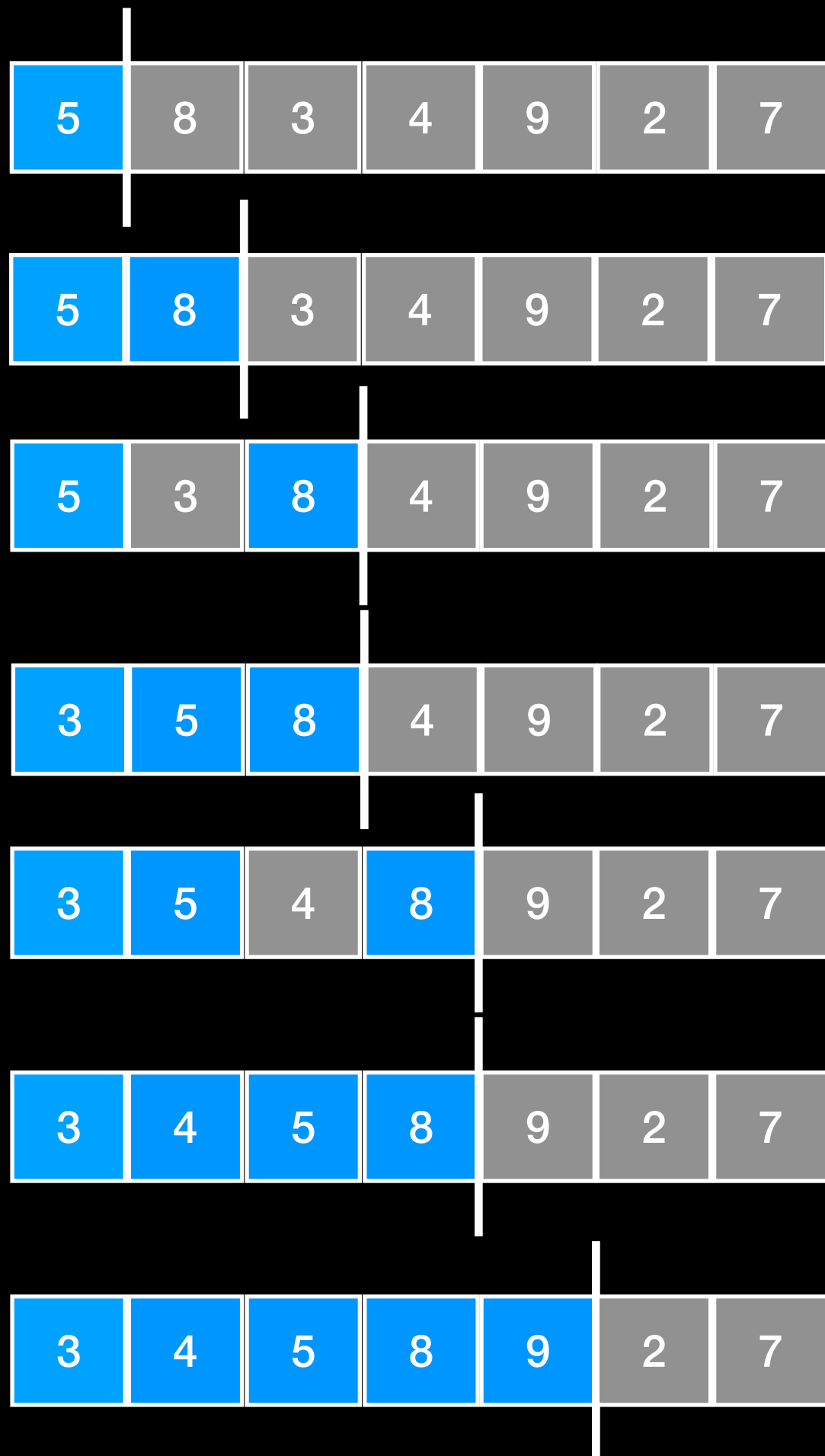
Pick first element in
unsorted region and put it in
right place in sorted region

Lecture Activity

















Sort the array using **Insertion Sort**

Show the entire array after each comparison/swap operation and at each step mark clearly the division between the sorted and unsorted portions of the array





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

 Play All	 Insertion	 Selection	 Bubble
 Random			
 Nearly Sorted			
 Reversed			

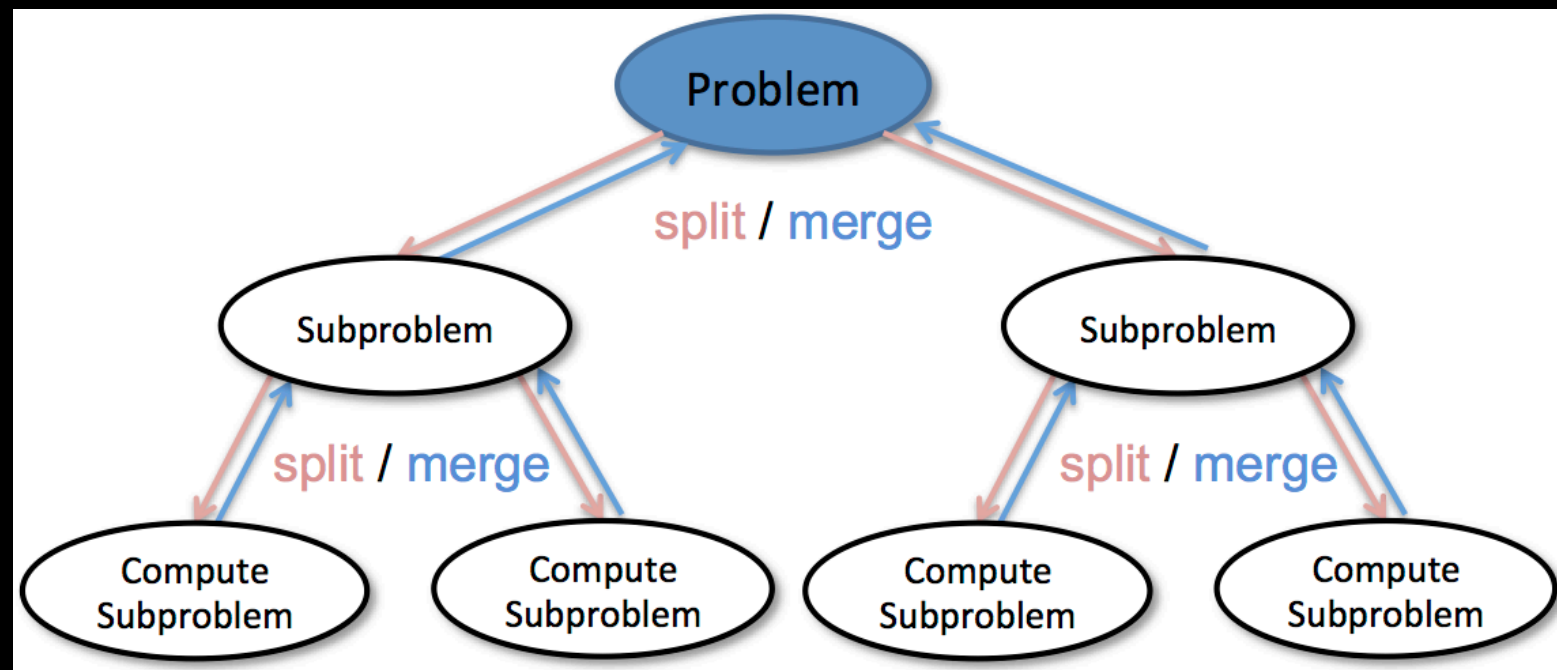
What we have so far

	Worst Case	Best Case
Selection Sort	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n)$

Can we do better?

Can we do better?

Divide and Conquer!!!



Merge Sort

Understanding $O(n^2)$

100	14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932	5
-----	----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----	---

$T(n)$

Understanding $O(n^2)$

100	14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932	5
-----	----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----	---

$T(n)$

100	14	3	43	200	274	523	108	76
-----	----	---	----	-----	-----	-----	-----	----

195	599	158	2	260	11	64	932	5
-----	-----	-----	---	-----	----	----	-----	---

Understanding $O(n^2)$

100	14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932	5
-----	----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----	---

$T(n)$

100	14	3	43	200	274	523	108	76
-----	----	---	----	-----	-----	-----	-----	----

$T(\frac{1}{2}n)$

195	599	158	2	260	11	64	932	5
-----	-----	-----	---	-----	----	----	-----	---

$T(\frac{1}{2}n)$

Understanding $O(n^2)$

100	14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932	5
-----	----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----	---

$T(n)$

100	14	3	43	200	274	523	108	76
-----	----	---	----	-----	-----	-----	-----	----

$T(\frac{1}{2}n)$

195	599	158	2	260	11	64	932	5
-----	-----	-----	---	-----	----	----	-----	---

$T(\frac{1}{2}n)$

$$(n/2)^2 = n^2/4$$

Understanding $O(n^2)$

100	14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932	5
-----	----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----	---

$T(n)$

100	14	3	43	200	274	523	108	76
-----	----	---	----	-----	-----	-----	-----	----

195	599	158	2	260	11	64	932	5
-----	-----	-----	---	-----	----	----	-----	---

$$T(1/2n) \approx 1/4 T(n)$$

$$T(1/2n) \approx 1/4 T(n)$$

$$(n/2)^2 = n^2/4$$

Understanding $O(n^2)$

100	14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932	5
-----	----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----	---

$T(n)$

3	14	43	76	100	108	200	274	523
---	----	----	----	-----	-----	-----	-----	-----

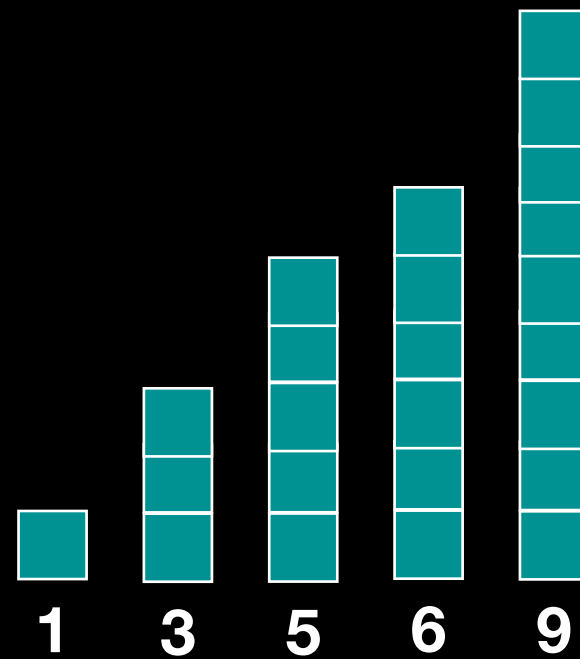
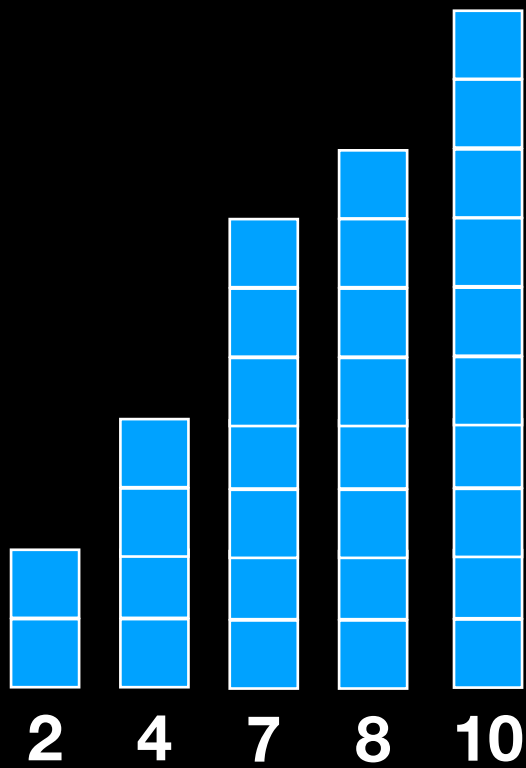
2	5	11	64	158	195	260	599	932
---	---	----	----	-----	-----	-----	-----	-----

$$T(1/2n) \approx 1/4 T(n)$$

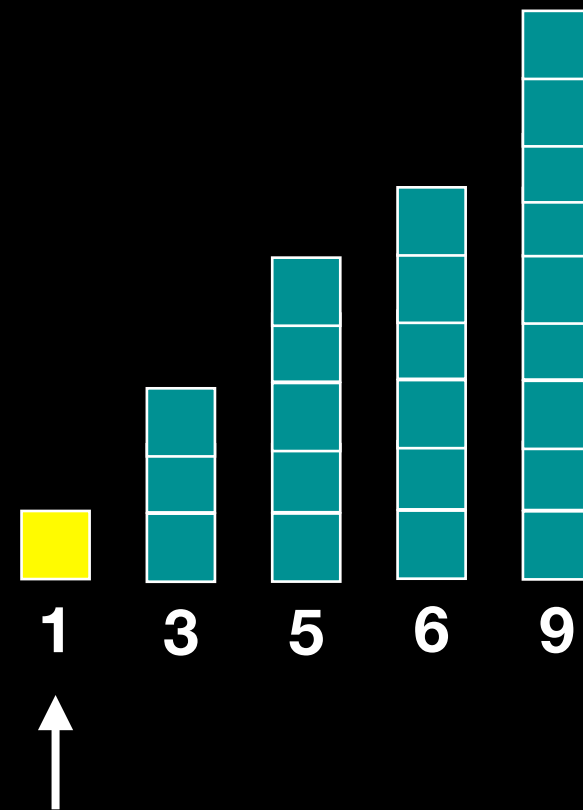
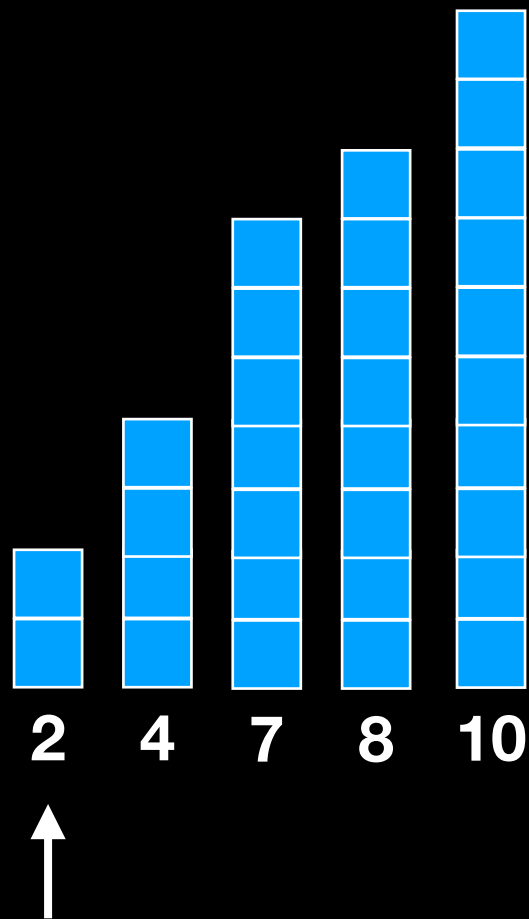
$$T(1/2n) \approx 1/4 T(n)$$

$$(n/2)^2 = n^2/4$$

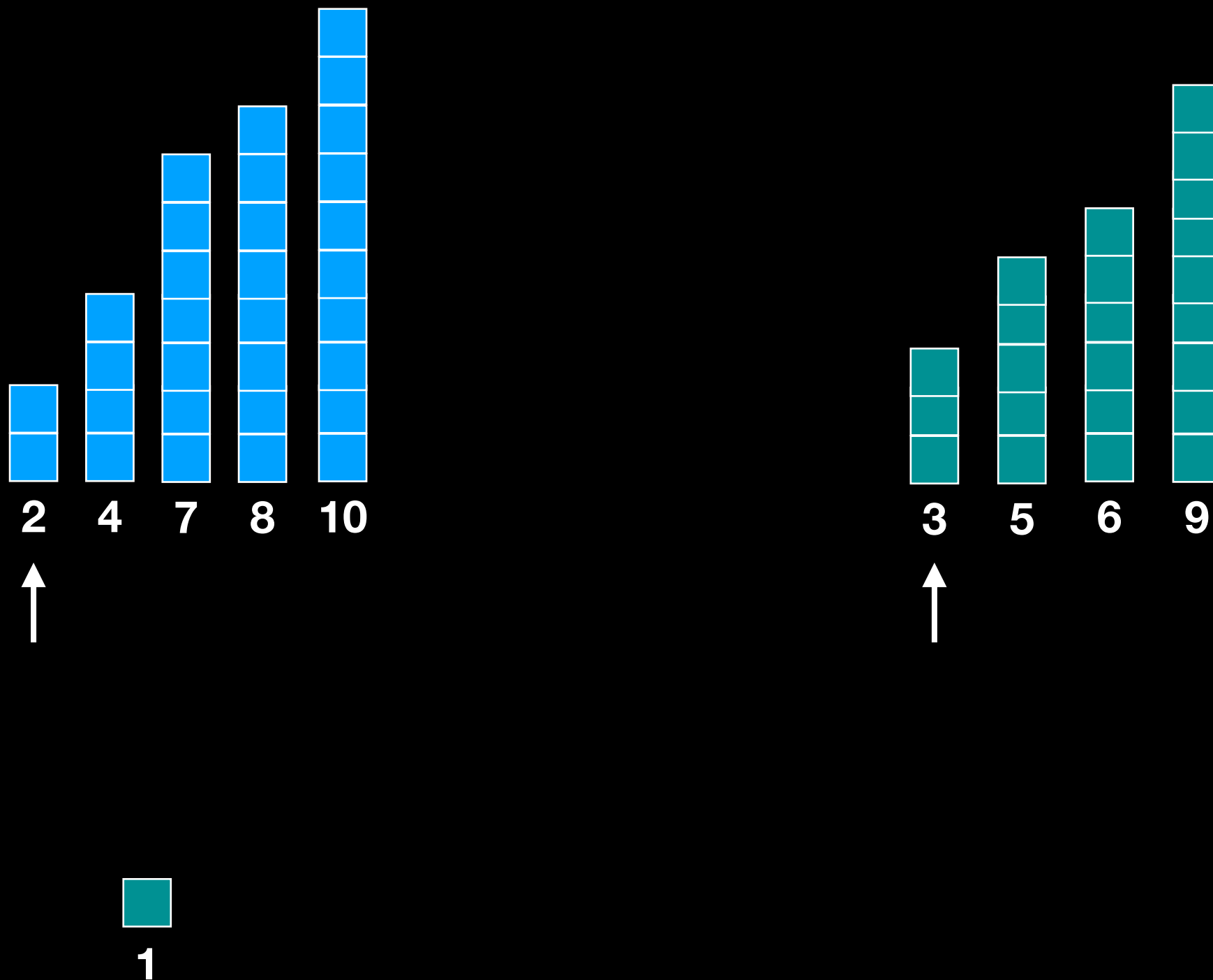
Key Insight: Merge is linear



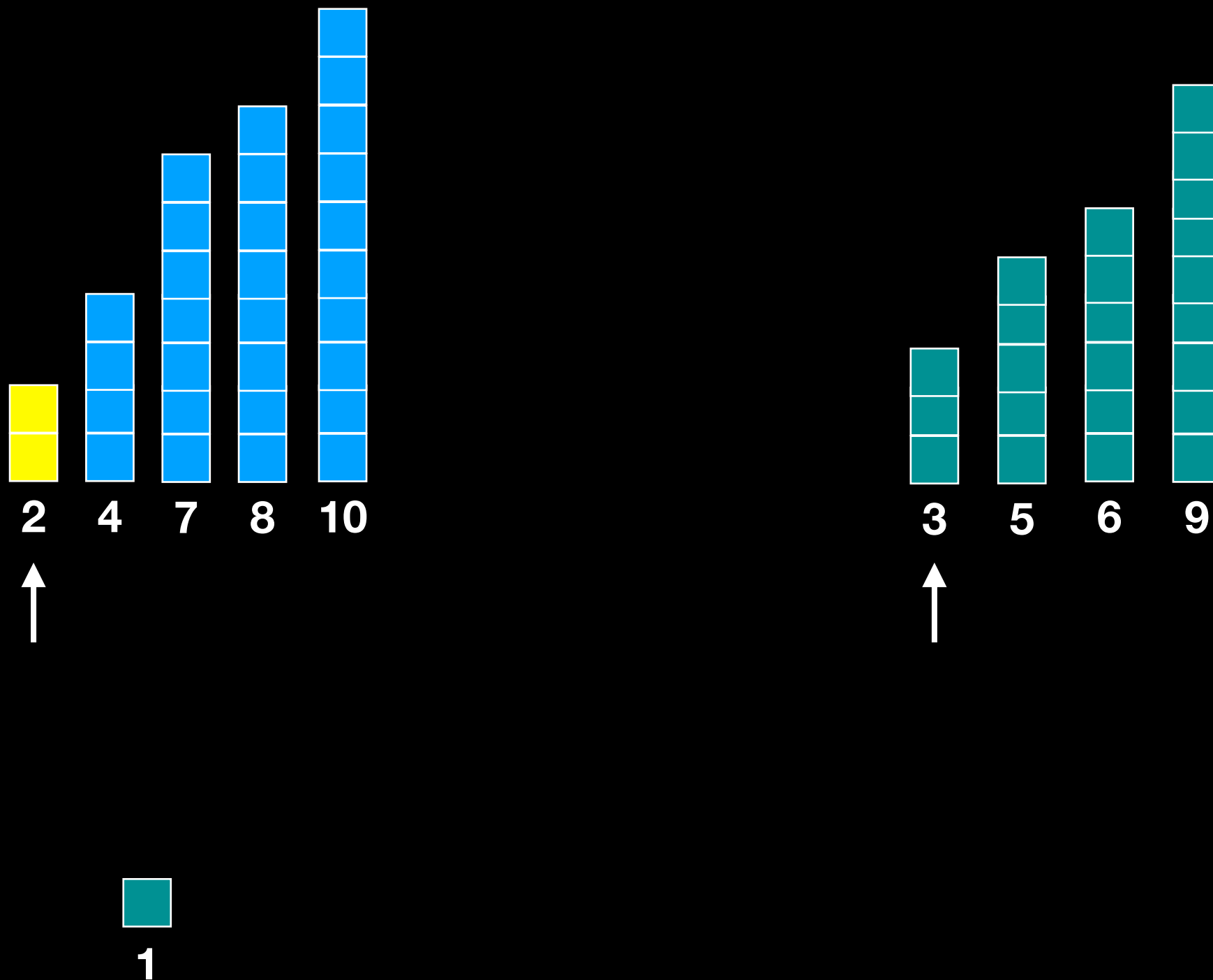
Key Insight: Merge is linear



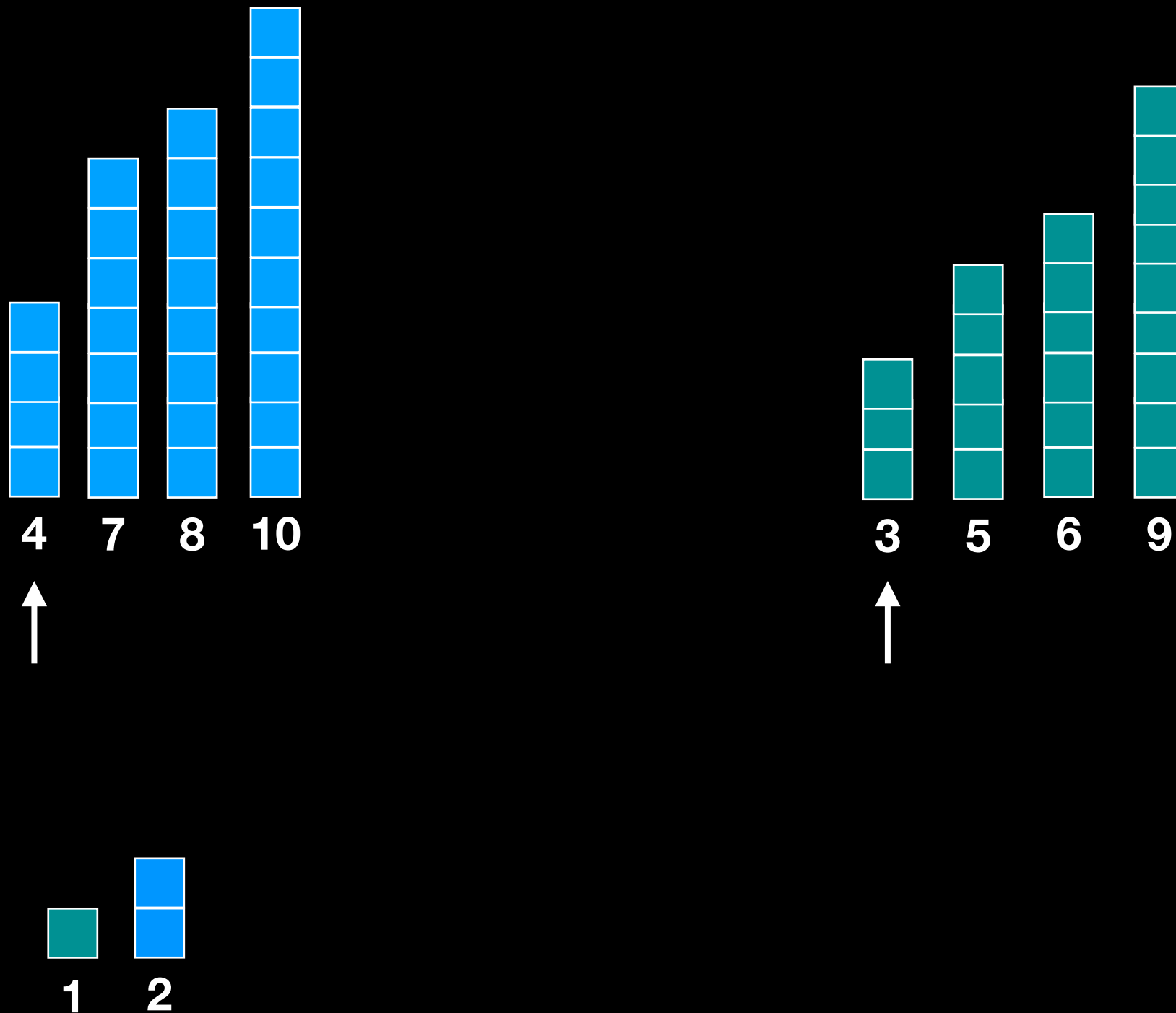
Key Insight: Merge is linear



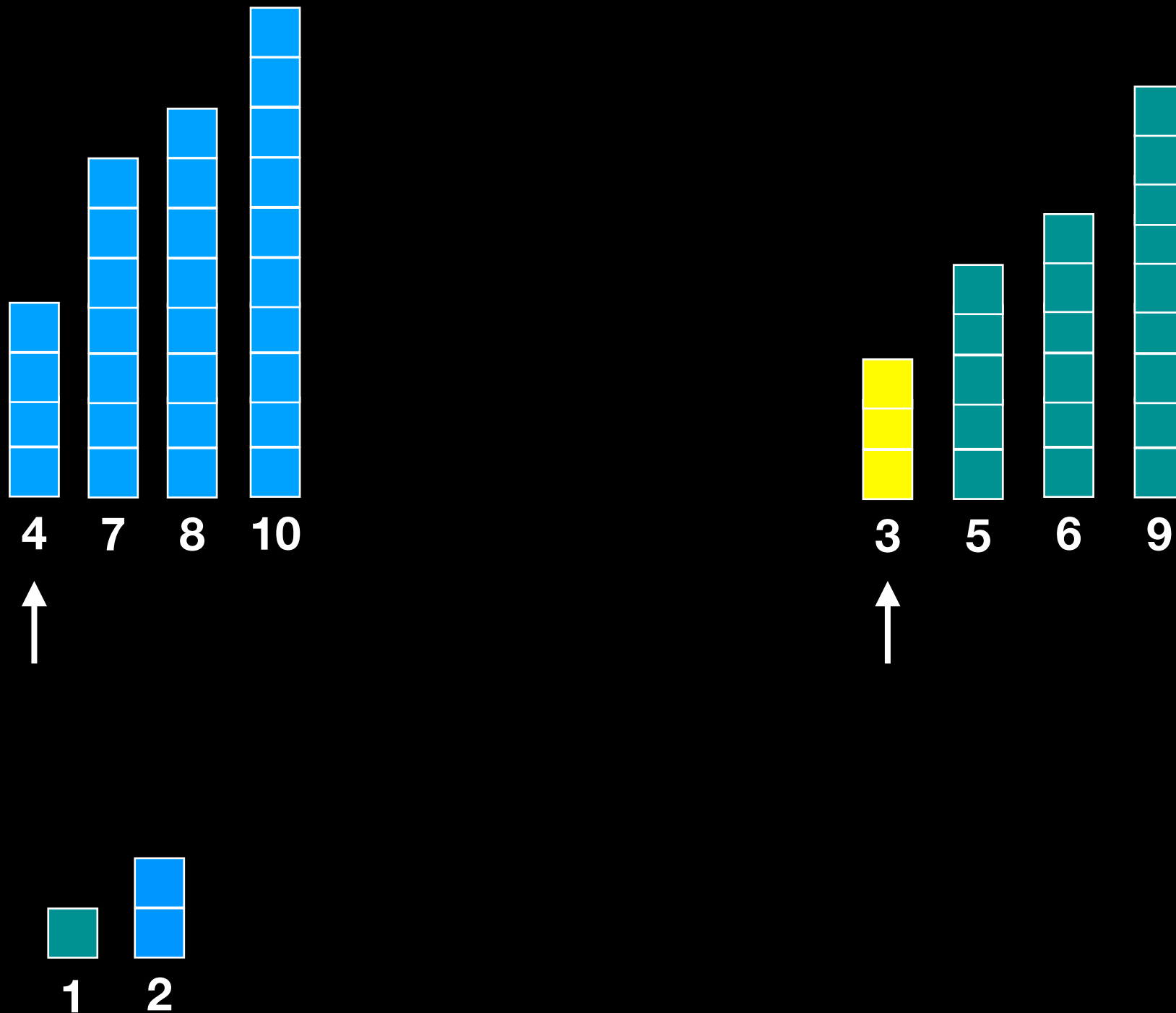
Key Insight: Merge is linear



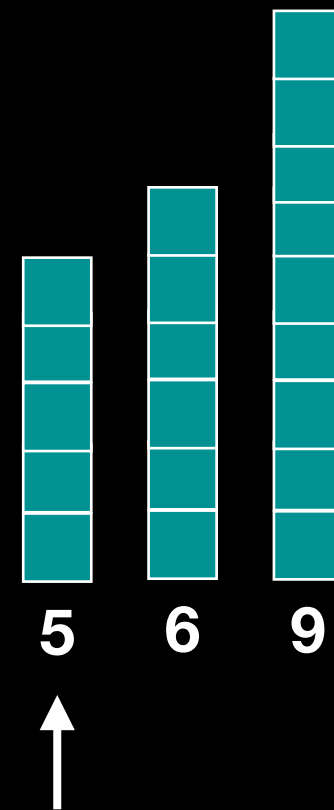
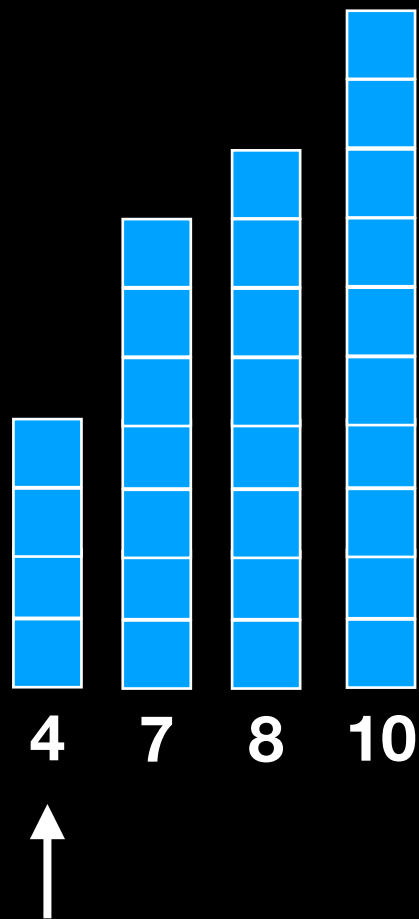
Key Insight: Merge is linear



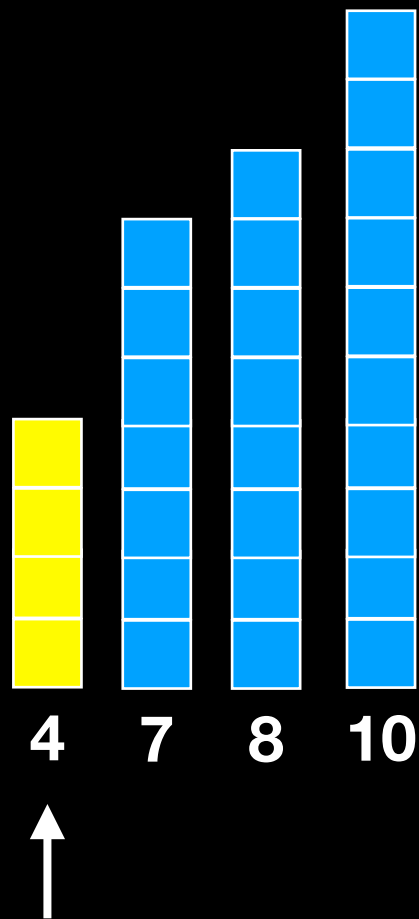
Key Insight: Merge is linear



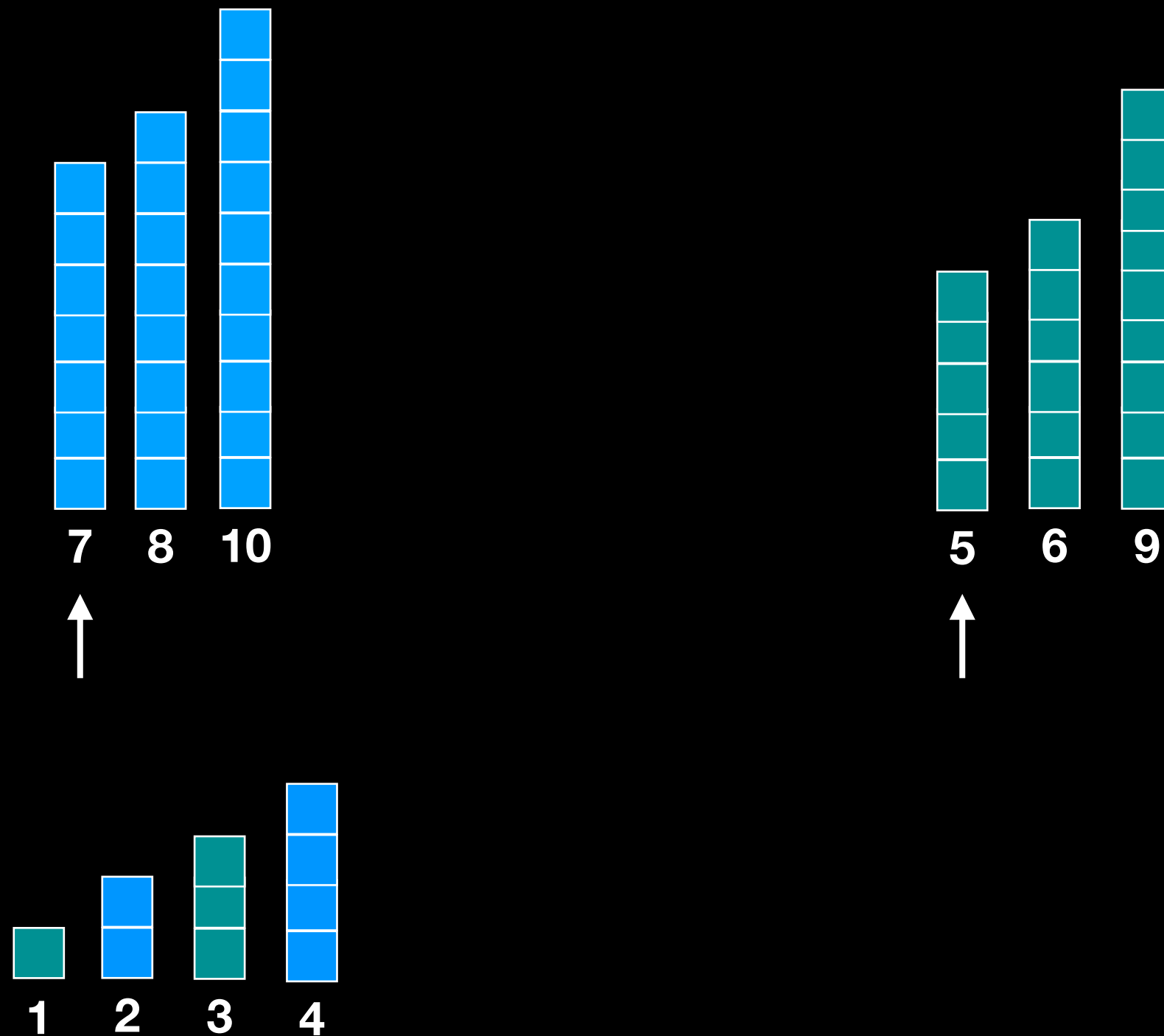
Key Insight: Merge is linear



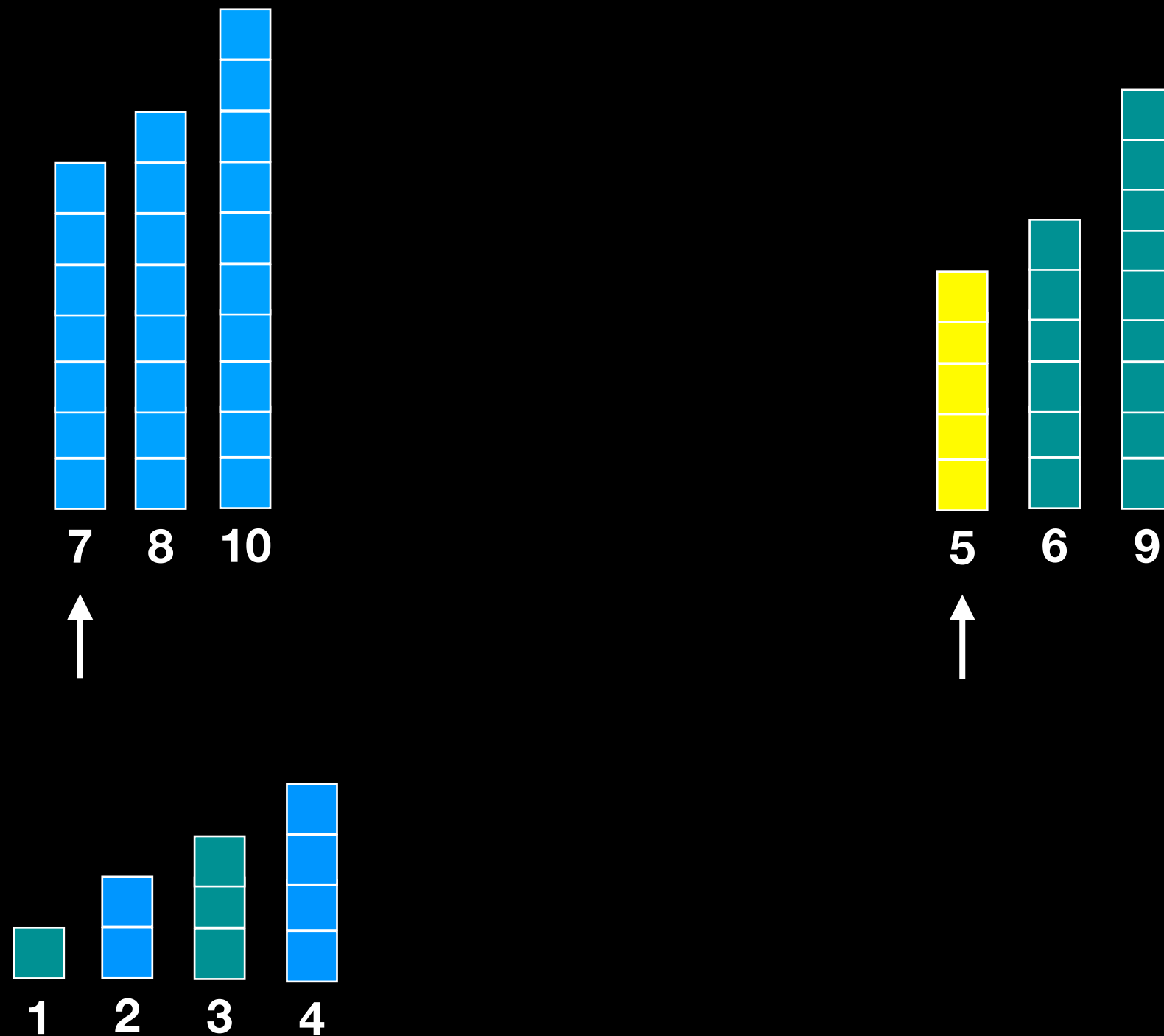
Key Insight: Merge is linear



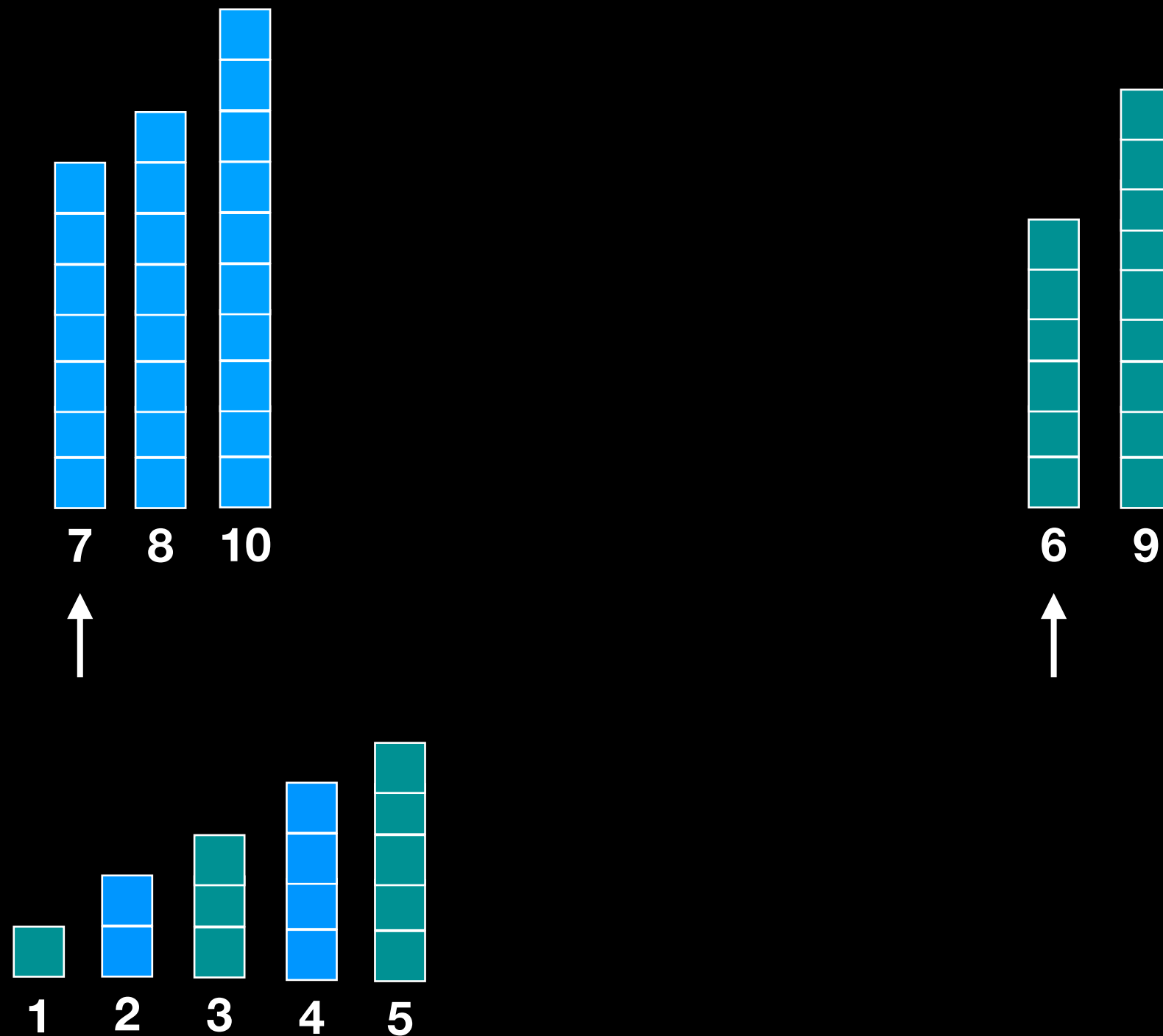
Key Insight: Merge is linear



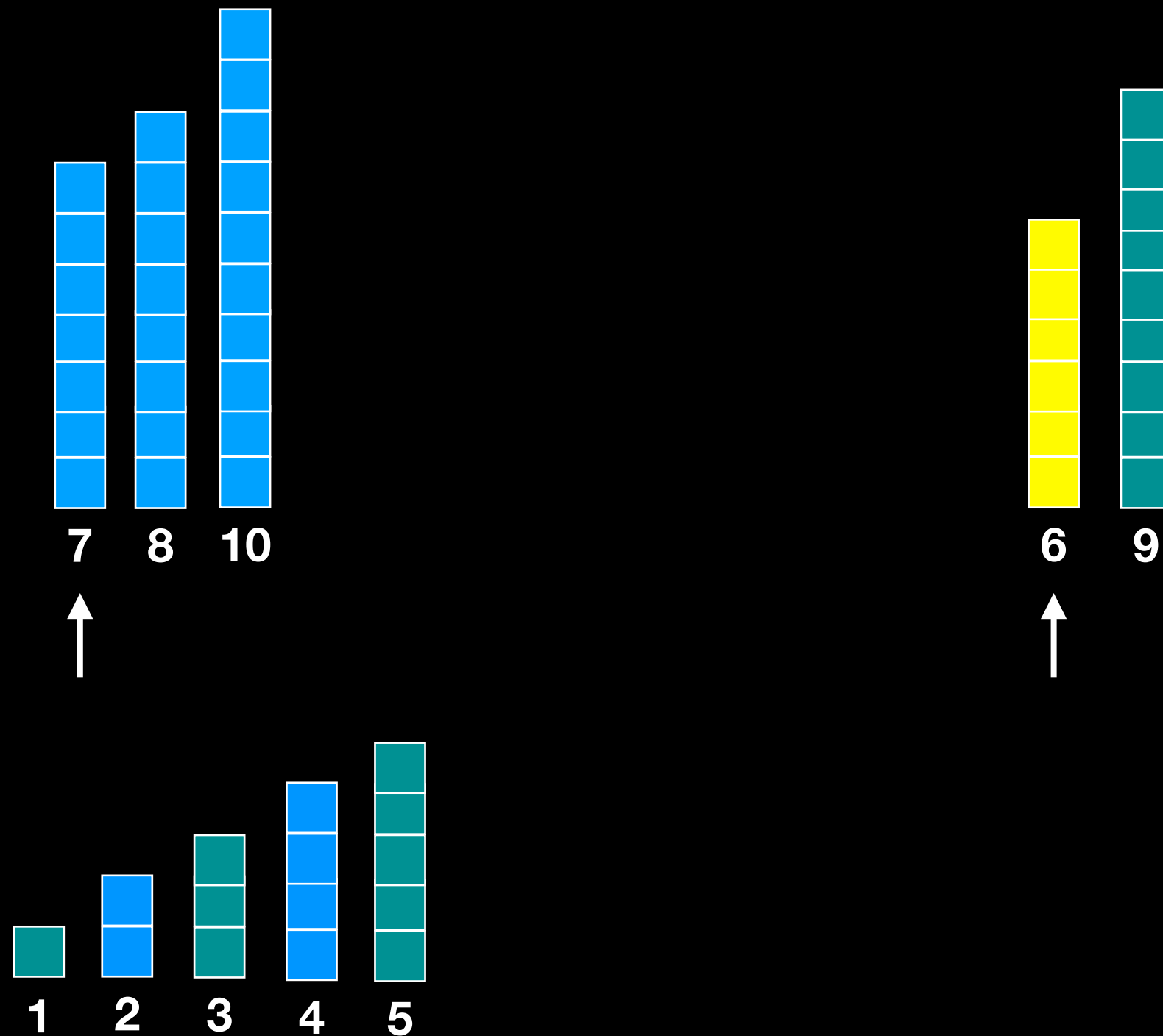
Key Insight: Merge is linear



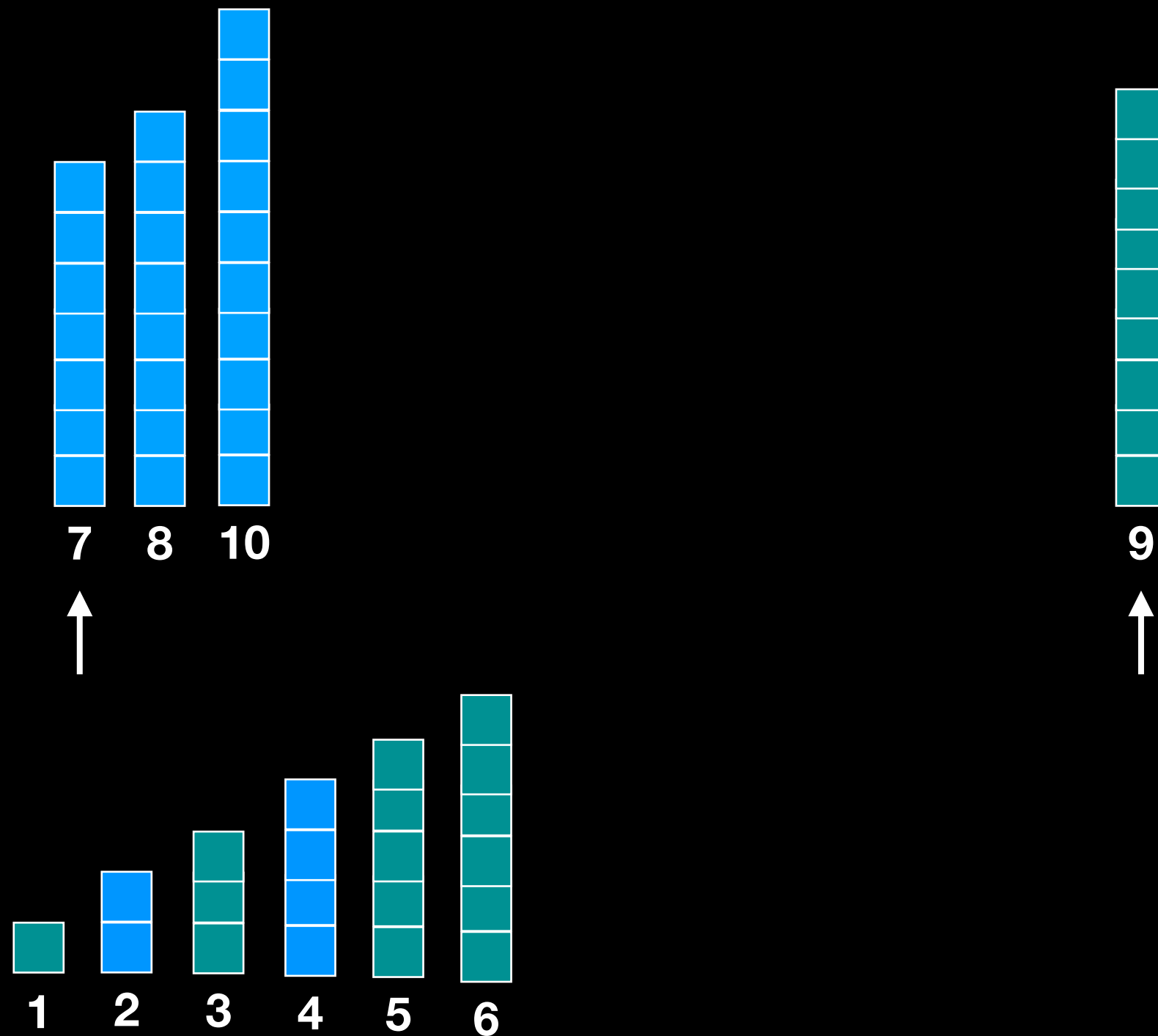
Key Insight: Merge is linear



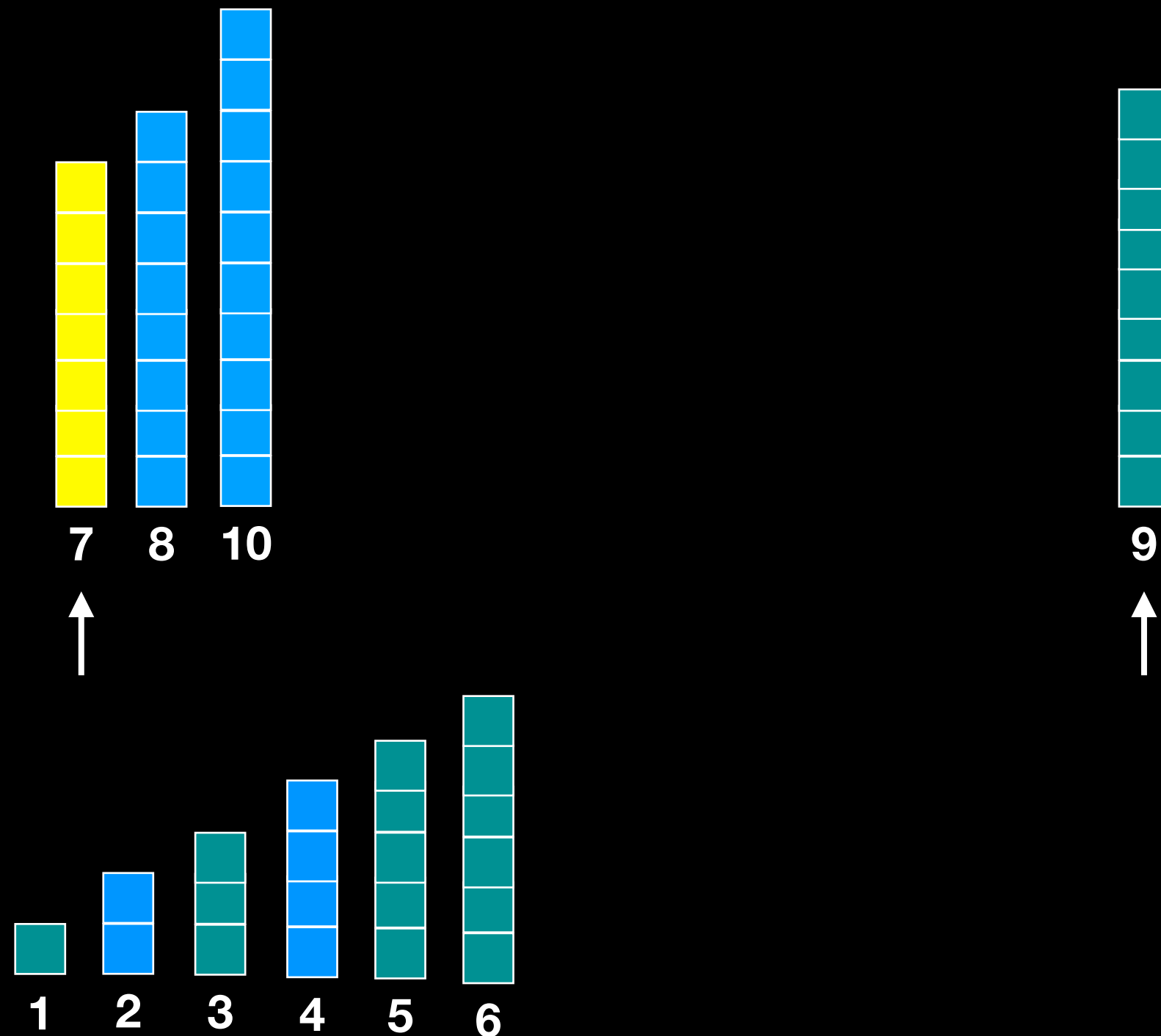
Key Insight: Merge is linear



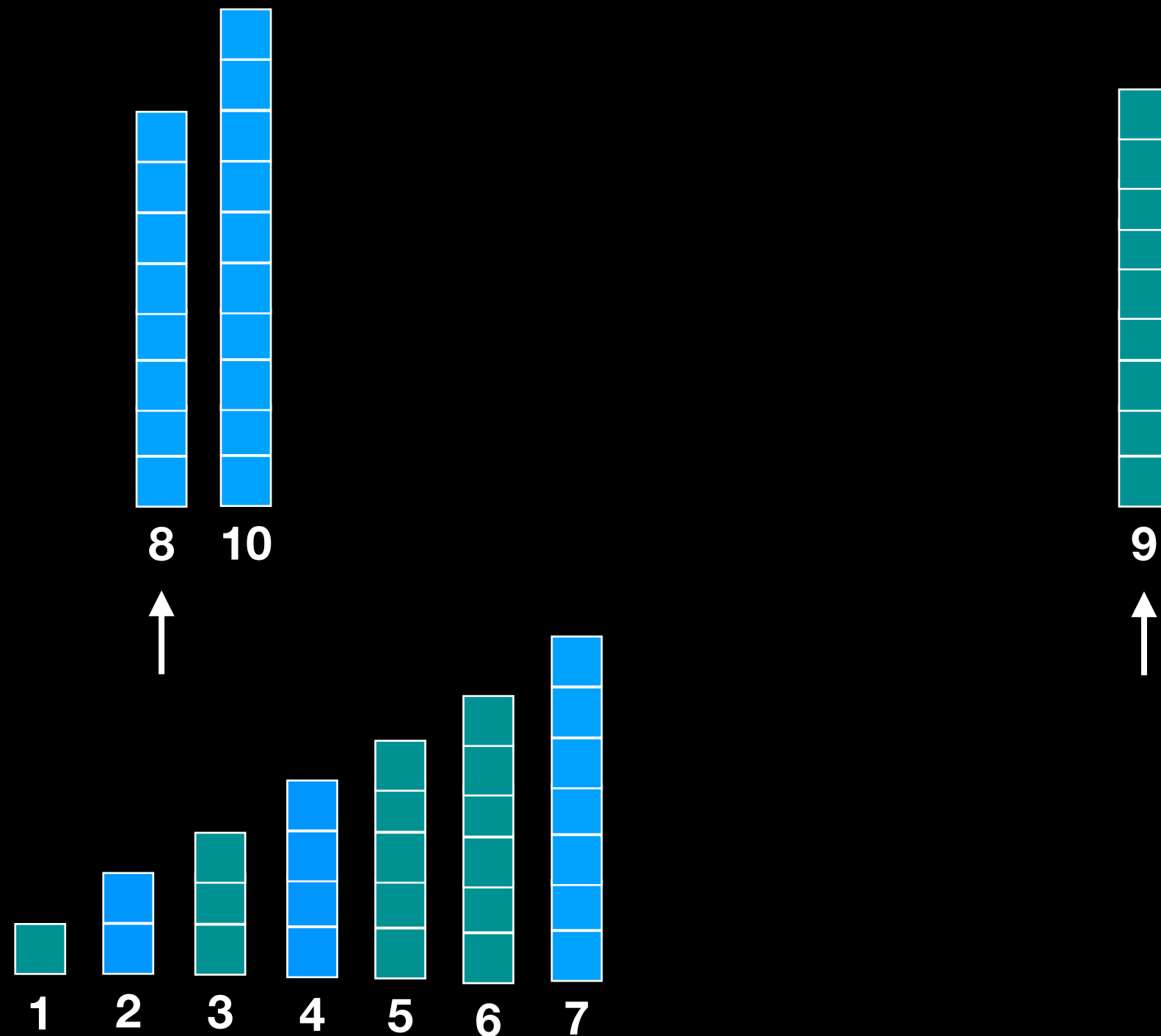
Key Insight: Merge is linear



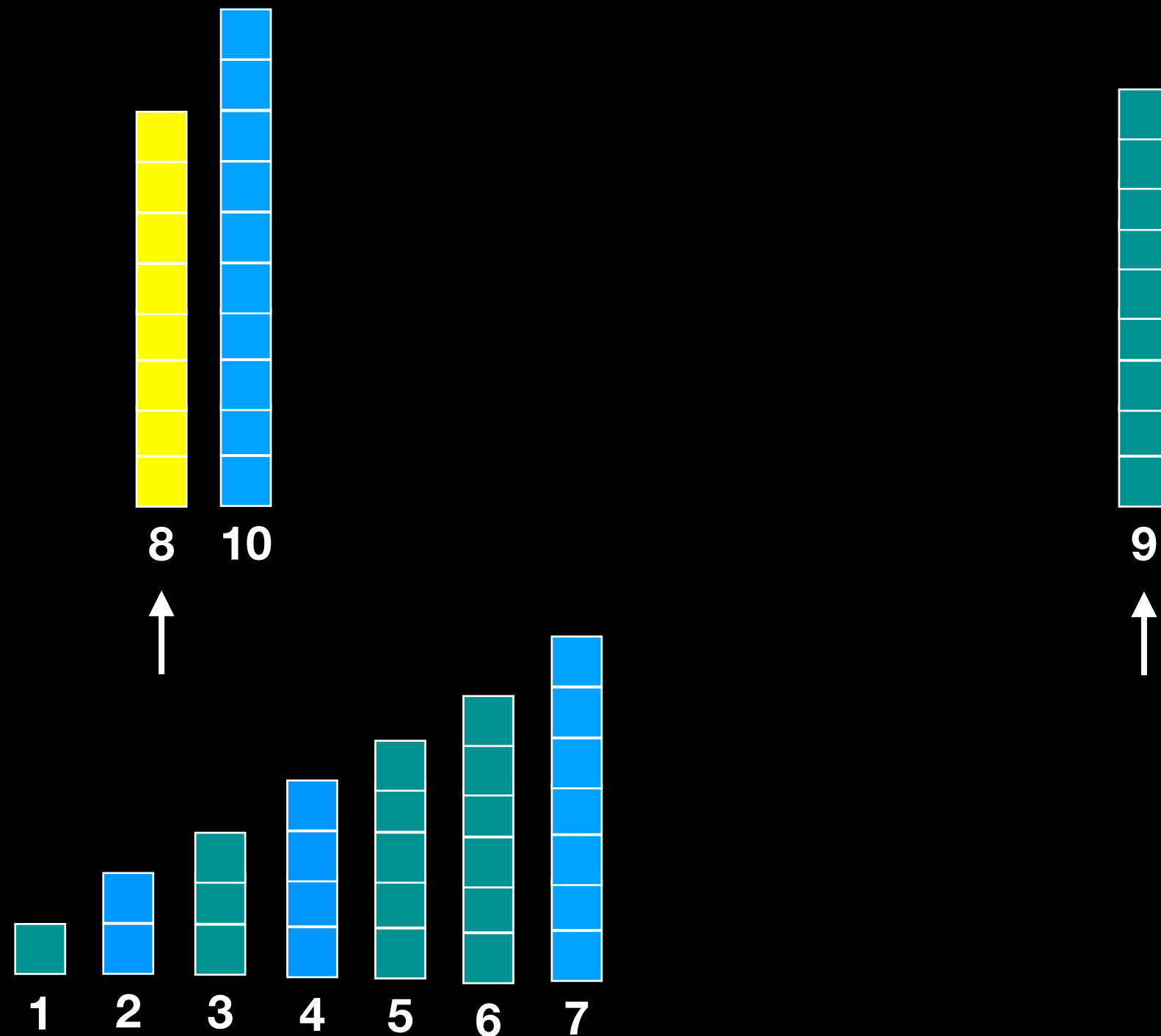
Key Insight: Merge is linear



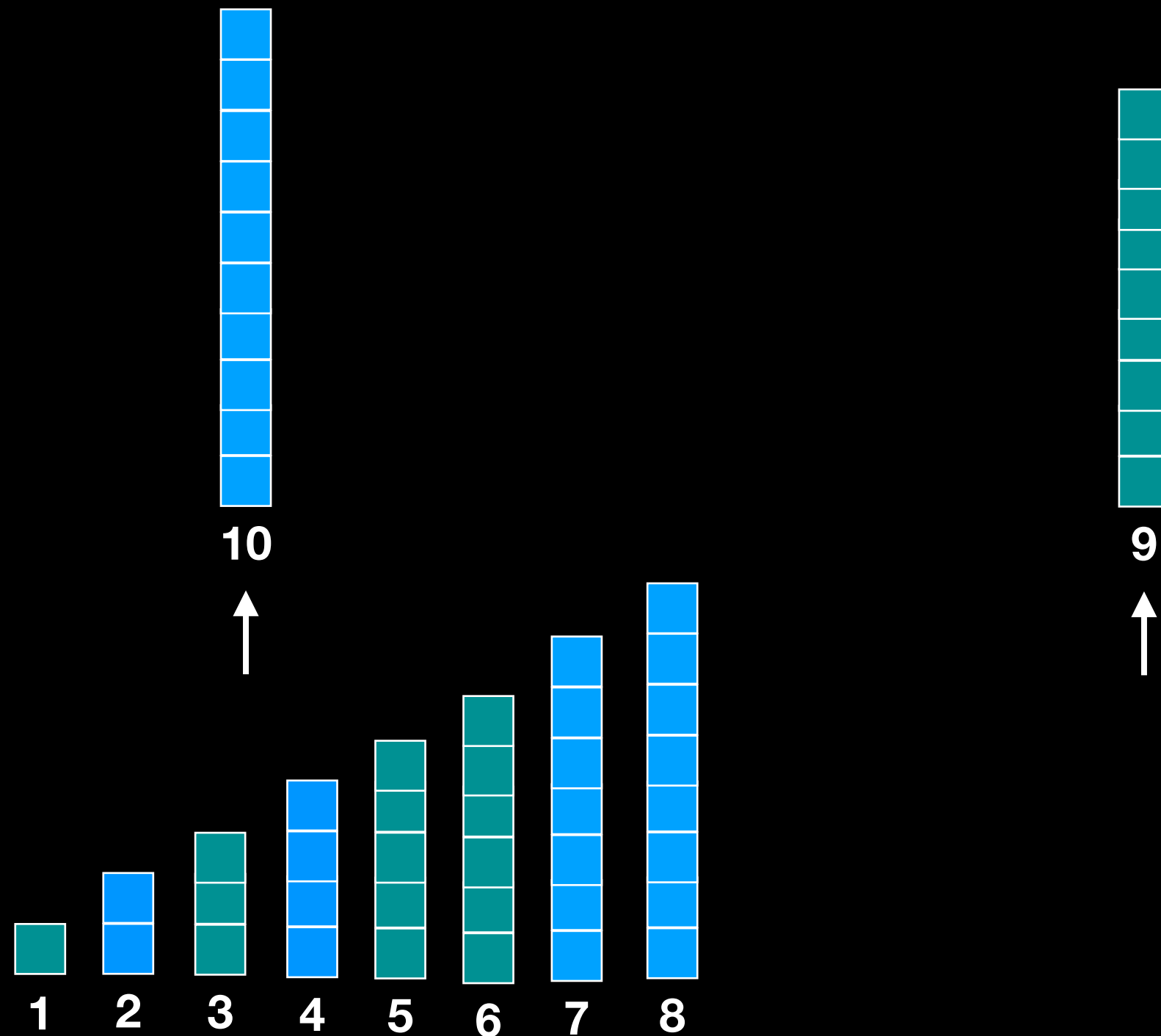
Key Insight: Merge is linear



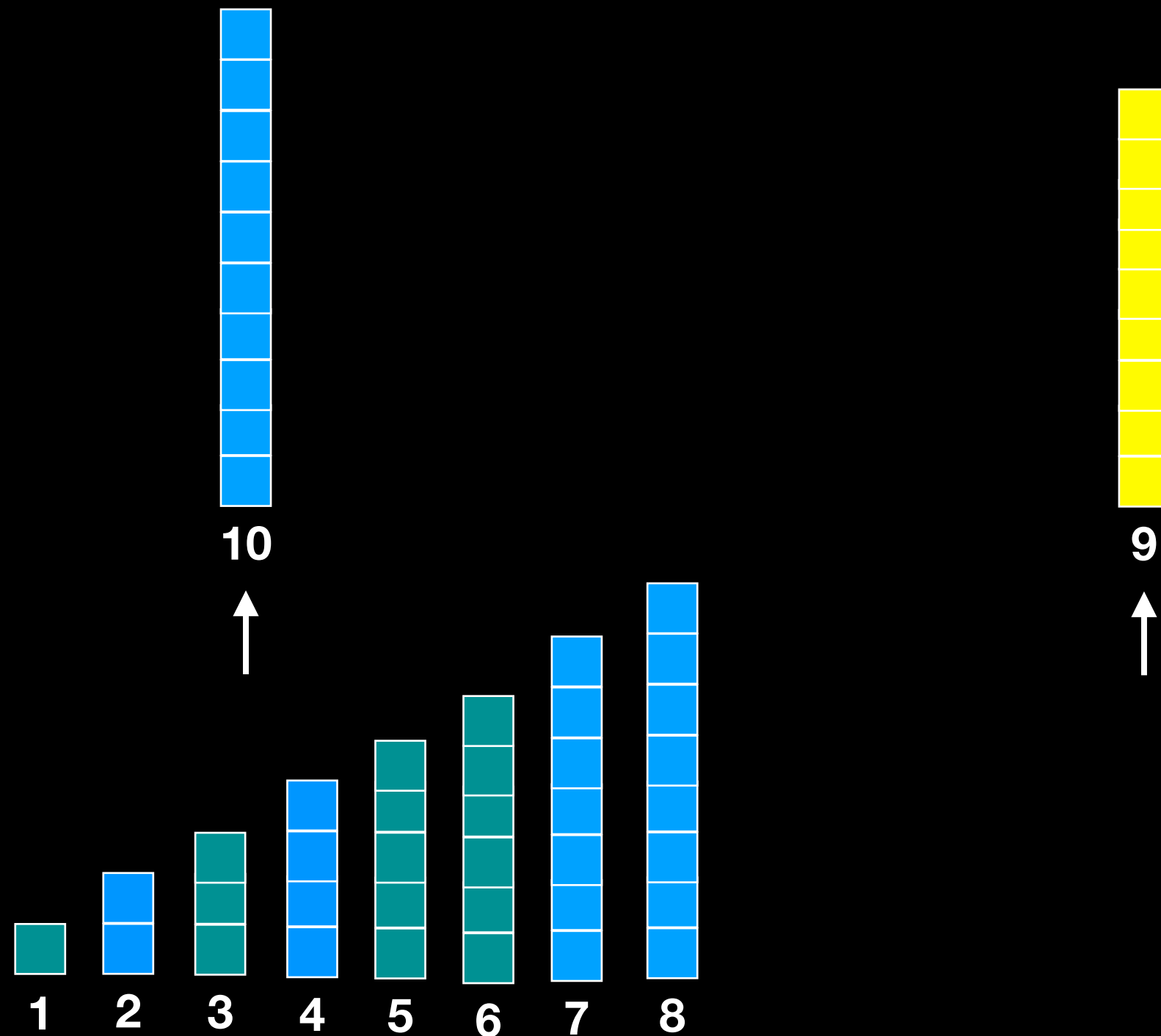
Key Insight: Merge is linear



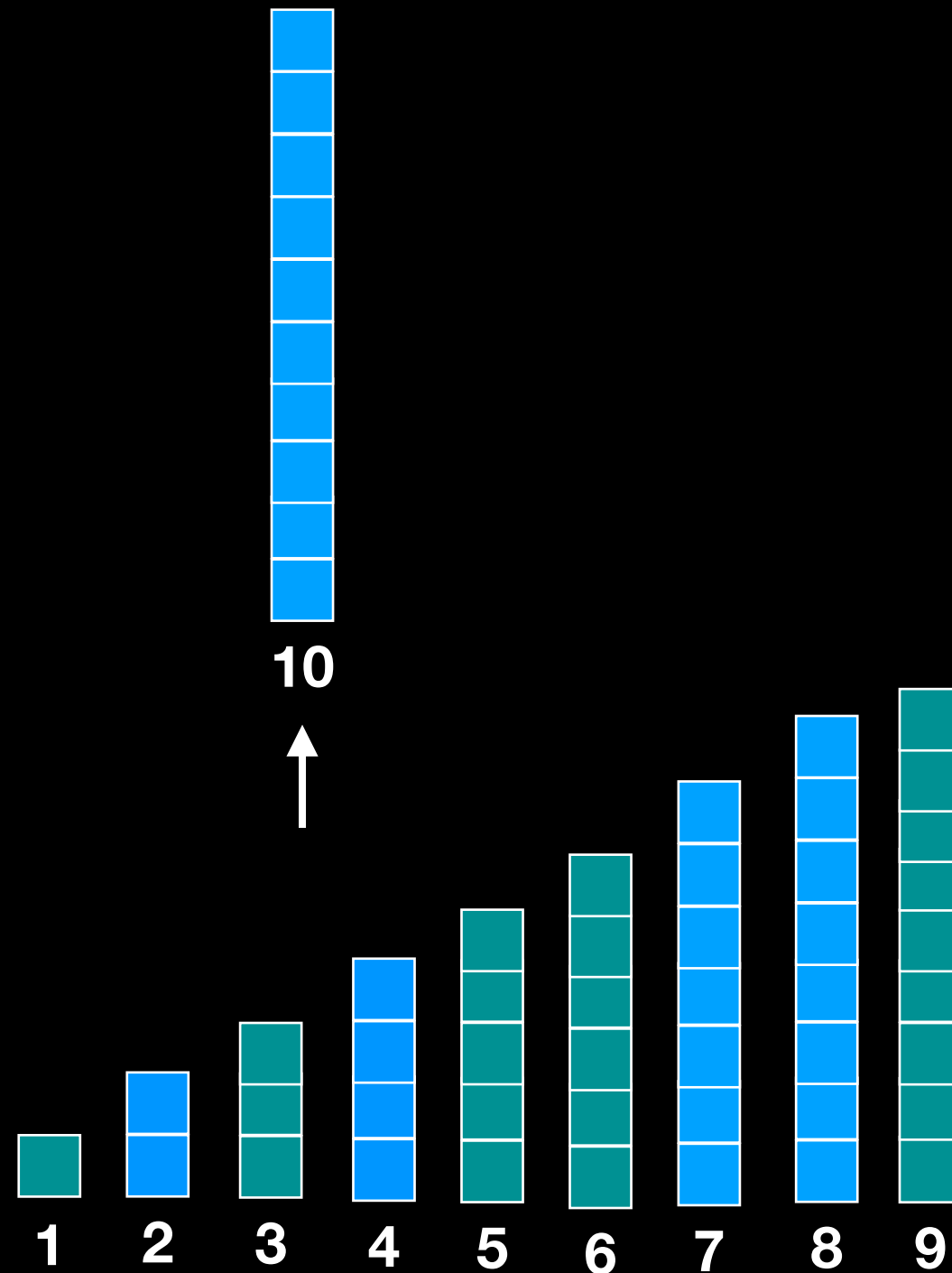
Key Insight: Merge is linear



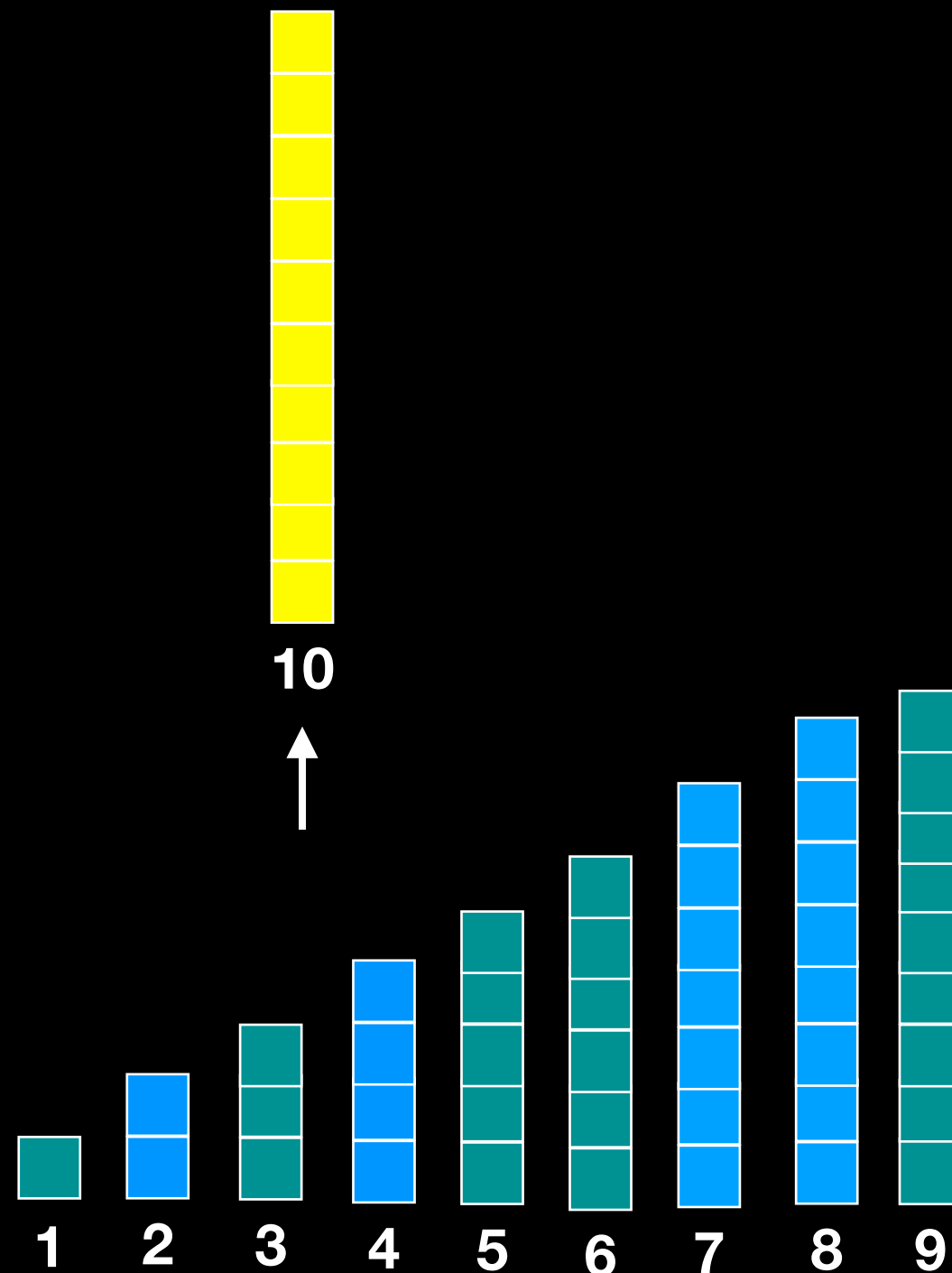
Key Insight: Merge is linear



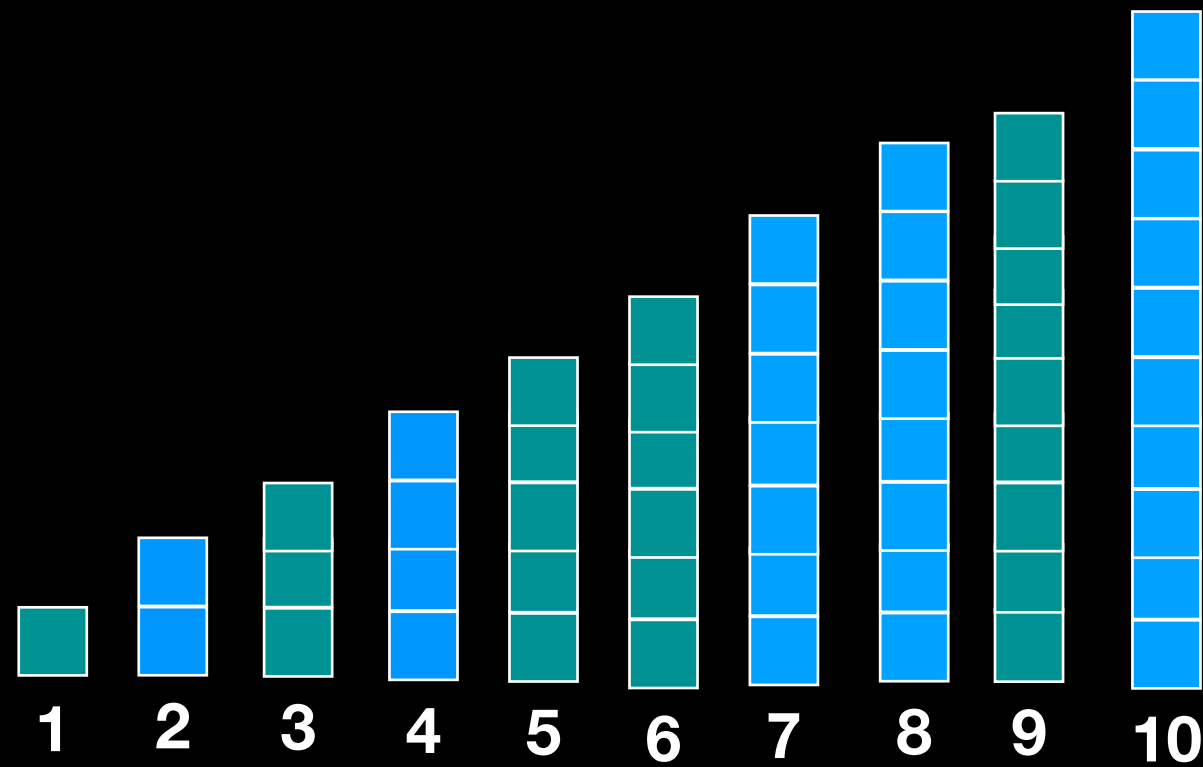
Key Insight: Merge is linear



Key Insight: Merge is linear



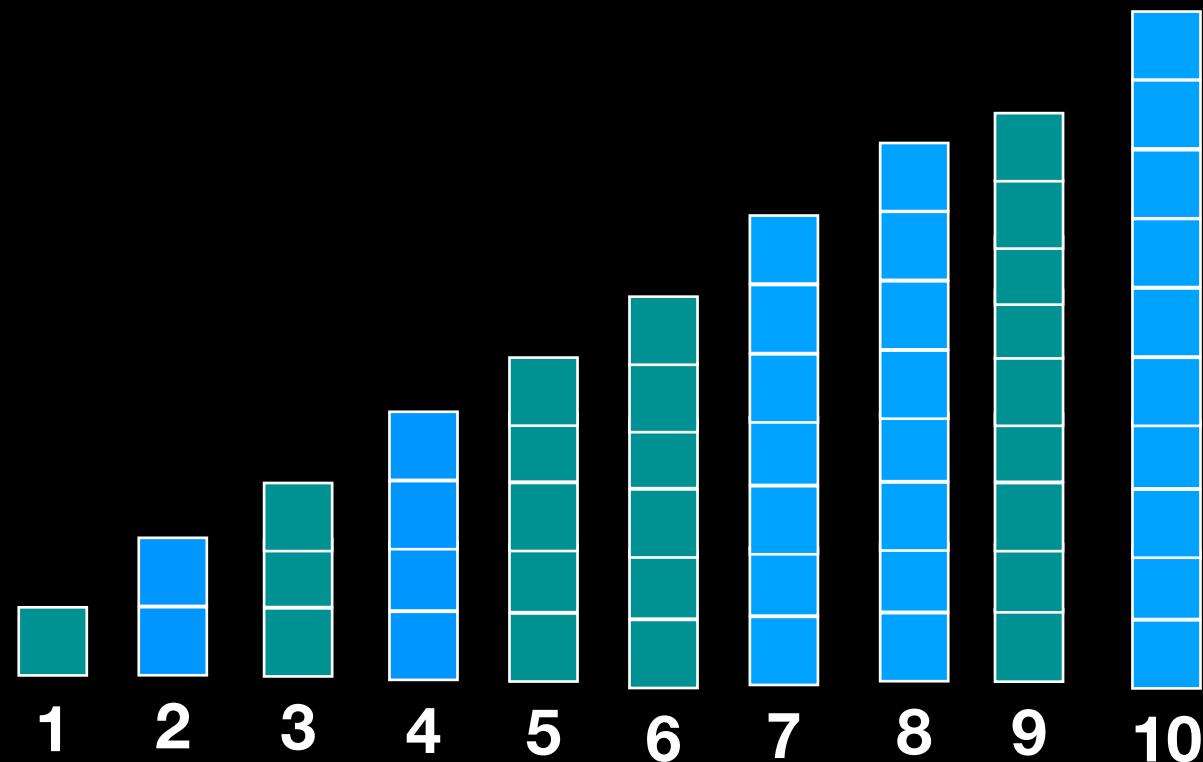
Key Insight: Merge is linear



Key Insight: Merge is linear

Each step makes one comparison and reduces the number of elements to be merged by 1.

If there are n total elements to be merged, merging is $O(n)$



Divide and Conquer

100	14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932	5
-----	----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----	---

$T(n)$

14	43	76	100	108	200	274	523
----	----	----	-----	-----	-----	-----	-----

$T(1/2n) \approx 1/4 T(n)$

11	64	158	195	260	599	932
----	----	-----	-----	-----	-----	-----

$T(1/2n) \approx 1/4 T(n)$

Divide and Conquer

100	14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932	5
-----	----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----	---

$T(n)$

14	43	76	100	108	200	274	523
----	----	----	-----	-----	-----	-----	-----

11	64	158	195	260	599	932
----	----	-----	-----	-----	-----	-----

$T(1/2n) \approx 1/4 T(n)$

$T(1/2n) \approx 1/4 T(n)$

2	3	5	11	14	43	64	76	100	108	158	195	200	260	274	523	599	932
---	---	---	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

$T(n) \approx 1/2 T(n) + n$

Speed up insertion sort by a **factor of two** by splitting in half, sorting separately and merging results!

Divide and Conquer

Splitting in two gives 2x improvement.

Divide and Conquer

Splitting in two gives 2x improvement.

Splitting in four gives 4x improvement.

Divide and Conquer

Splitting in two gives 2x improvement.

Splitting in four gives 4x improvement.

Splitting in eight gives 8x improvement.

Divide and Conquer

Splitting in two gives 2x improvement.

Splitting in four gives 4x improvement.

Splitting in eight gives 8x improvement.

What if we never stop splitting?

Merge Sort

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

14	3	43	200	274	523	108	76
----	---	----	-----	-----	-----	-----	----

195	599	158	2	260	11	64	932
-----	-----	-----	---	-----	----	----	-----

14	3	43	200
----	---	----	-----

274	523	108	76
-----	-----	-----	----

195	599	158	2
-----	-----	-----	---

260	11	64	932
-----	----	----	-----

14	3
----	---

43	200
----	-----

274	523
-----	-----

108	76
-----	----

195	599
-----	-----

158	2
-----	---

260	11
-----	----

64	932
----	-----

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

Merge Sort

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

14	3	43	200	274	523	108	76
----	---	----	-----	-----	-----	-----	----

195	599	158	2	260	11	64	932
-----	-----	-----	---	-----	----	----	-----

14	3	43	200
----	---	----	-----

274	523	108	76
-----	-----	-----	----

195	599	158	2
-----	-----	-----	---

260	11	64	932
-----	----	----	-----

14	3
----	---

43	200
----	-----

274	523
-----	-----

108	76
-----	----

195	599
-----	-----

158	2
-----	---

260	11
-----	----

64	932
----	-----

14

3

43

200

274

523

108

76

195

599

158

2

260

11

64

932

Merge Sort

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

14	3	43	200	274	523	108	76
----	---	----	-----	-----	-----	-----	----

195	599	158	2	260	11	64	932
-----	-----	-----	---	-----	----	----	-----

14	3	43	200
----	---	----	-----

274	523	108	76
-----	-----	-----	----

195	599	158	2
-----	-----	-----	---

260	11	64	932
-----	----	----	-----

3	14
---	----

43	200
----	-----

274	523
-----	-----

76	108
----	-----

195	599
-----	-----

2	158
---	-----

11	26
----	----

64	932
----	-----

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

Merge Sort

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

14	3	43	200	274	523	108	76
----	---	----	-----	-----	-----	-----	----

195	599	158	2	260	11	64	932
-----	-----	-----	---	-----	----	----	-----

3	14	43	200
---	----	----	-----

76	108	274	523
----	-----	-----	-----

2	158	195	599
---	-----	-----	-----

11	26	64	932
----	----	----	-----

3	14
---	----

43	200
----	-----

274	523
-----	-----

76	108
----	-----

195	599
-----	-----

2	158
---	-----

11	26
----	----

64	932
----	-----

14

3

43

200

274

523

108

76

195

599

158

2

260

11

64

932

Merge Sort

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

3	14	43	76	108	200	274	523
---	----	----	----	-----	-----	-----	-----

2	11	26	64	158	195	599	932
---	----	----	----	-----	-----	-----	-----

3	14	43	200
---	----	----	-----

76	108	274	523
----	-----	-----	-----

2	158	195	599
---	-----	-----	-----

11	26	64	932
----	----	----	-----

3	14
---	----

43	200
----	-----

274	523
-----	-----

76	108
----	-----

195	599
-----	-----

2	158
---	-----

11	26
----	----

64	932
----	-----

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

Merge Sort

2	3	11	14	26	43	64	76	108	158	195	200	274	523	599	932
---	---	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----

3	14	43	76	108	200	274	523
---	----	----	----	-----	-----	-----	-----

2	11	26	64	158	195	599	932
---	----	----	----	-----	-----	-----	-----

3	14	43	200
---	----	----	-----

76	108	274	523
----	-----	-----	-----

2	158	195	599
---	-----	-----	-----

11	26	64	932
----	----	----	-----

3	14
---	----

43	200
----	-----

274	523
-----	-----

76	108
----	-----

195	599
-----	-----

2	158
---	-----

11	26
----	----

64	932
----	-----

14

3

43

200

274

523

108

76

195

599

158

2

260

11

64

932

Merge Sort

2	3	11	14	26	43	64	76	108	158	195	200	274	523	599	932
---	---	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----

3	14	43	76	108	200	274	523
---	----	----	----	-----	-----	-----	-----

2	11	26	64	158	195	599	932
---	----	----	----	-----	-----	-----	-----

3	14	43	200
---	----	----	-----

76	108	274	523
----	-----	-----	-----

2	158	195	599
---	-----	-----	-----

11	26	64	932
----	----	----	-----

3	14
---	----

43	200
----	-----

274	523
-----	-----

76	108
----	-----

195	599
-----	-----

2	158
---	-----

11	26
----	----

64	932
----	-----

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

Merge Sort Analysis

2	3	11	14	26	43	64	76	108	158	195	200	274	523	599	932
---	---	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----

$O(n)$

3	14	43	76	108	200	274	523
---	----	----	----	-----	-----	-----	-----

2	11	26	64	158	195	599	932
---	----	----	----	-----	-----	-----	-----

$O(n)$

3	14	43	200
---	----	----	-----

76	108	274	523
----	-----	-----	-----

2	158	195	599
---	-----	-----	-----

11	26	64	932
----	----	----	-----

$O(n)$

3	14
---	----

43	200
----	-----

274	523
-----	-----

76	108
----	-----

195	599
-----	-----

2	158
---	-----

11	26
----	----

64	932
----	-----

$O(n)$

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

Merge Sort Analysis

2	3	11	14	26	43	64	76	108	158	195	200	274	523	599	932
---	---	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----

n

3	14	43	76	108	200	274	523
---	----	----	----	-----	-----	-----	-----

2	11	26	64	158	195	599	932
---	----	----	----	-----	-----	-----	-----

n/2

3	14	43	200
---	----	----	-----

76	108	274	523
----	-----	-----	-----

2	158	195	599
---	-----	-----	-----

11	26	64	932
----	----	----	-----

n/4

3	14
---	----

43	200
----	-----

274	523
-----	-----

76	108
----	-----

195	599
-----	-----

2	158
---	-----

11	26
----	----

64	932
----	-----

...

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

n/2^k

Merge n how many times?

Merge Sort Analysis

2	3	11	14	26	43	64	76	108	158	195	200	274	523	599	932
---	---	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----

n

3	14	43	76	108	200	274	523
---	----	----	----	-----	-----	-----	-----

2	11	26	64	158	195	599	932
---	----	----	----	-----	-----	-----	-----

n/2

3	14	43	200
---	----	----	-----

76	108	274	523
----	-----	-----	-----

2	158	195	599
---	-----	-----	-----

11	26	64	932
----	----	----	-----

n/4

3	14
---	----

43	200
----	-----

274	523
-----	-----

76	108
----	-----

195	599
-----	-----

2	158
---	-----

11	26
----	----

64	932
----	-----

...

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

n/2^k

Merge n how many times? $n/2^k = 1$

$$n = 2^k$$

$$\log_2 n = k$$

Merge Sort Analysis

2	3	11	14	26	43	64	76	108	158	195	200	274	523	599	932
---	---	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----

n

3	14	43	76	108	200	274	523
---	----	----	----	-----	-----	-----	-----

2	11	26	64	158	195	599	932
---	----	----	----	-----	-----	-----	-----

n/2

3	14	43	200
---	----	----	-----

76	108	274	523
----	-----	-----	-----

2	158	195	599
---	-----	-----	-----

11	26	64	932
----	----	----	-----

n/4

3	14
---	----

43	200
----	-----

274	523
-----	-----

76	108
----	-----

195	599
-----	-----

2	158
---	-----

11	26
----	----

64	932
----	-----

...

14

3

43

200

274

523

108

76

195

599

158

2

260

11

64

932

n/2^k

Merge n elements **log₂ n** times

Merge Sort Analysis

2	3	11	14	26	43	64	76	108	158	195	200	274	523	599	932
---	---	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----

$O(n)$

3	14	43	76	108	200	274	523
---	----	----	----	-----	-----	-----	-----

2	11	26	64	158	195	599	932
---	----	----	----	-----	-----	-----	-----

$O(n)$

3	14	43	200
---	----	----	-----

76	108	274	523
----	-----	-----	-----

2	158	195	599
---	-----	-----	-----

11	26	64	932
----	----	----	-----

$O(n)$

3	14
---	----

43	200
----	-----

274	523
-----	-----

76	108
----	-----

195	599
-----	-----

2	158
---	-----

11	26
----	----

64	932
----	-----

$O(n)$

14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932
----	---	----	-----	-----	-----	-----	----	-----	-----	-----	---	-----	----	----	-----

$O(n \log n)$

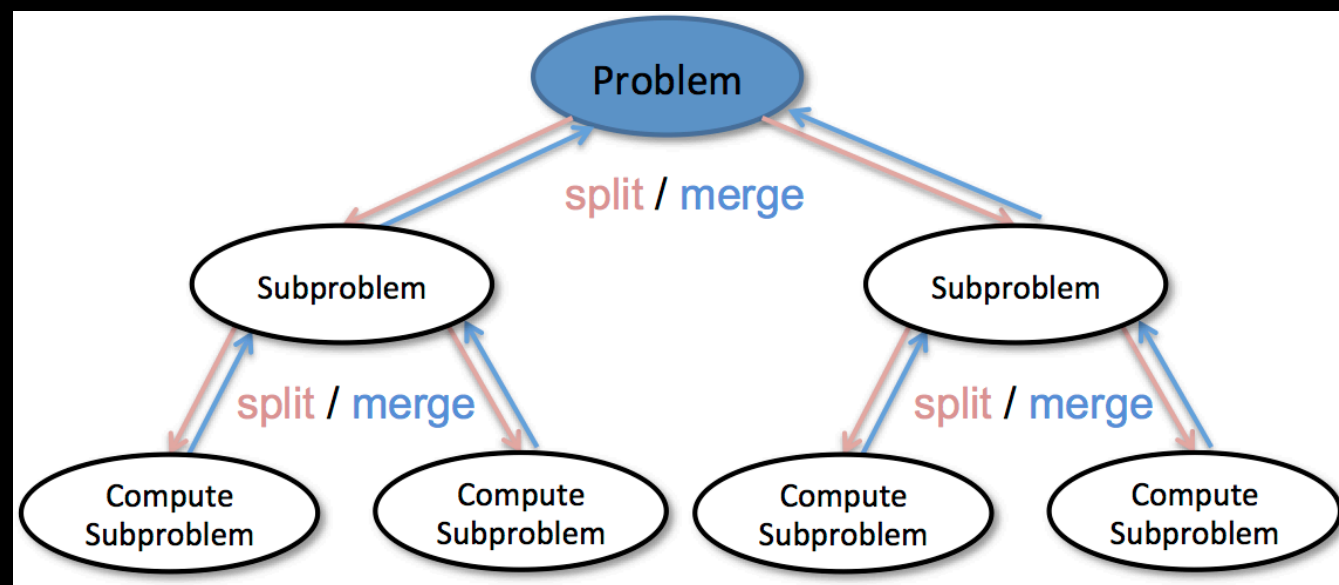
Merge Sort

How would you code this?

Merge Sort

How would you code this?

Hint: Divide and Conquer!!!



Merge Sort

```
void mergeSort(array)
{
    if array size <= 1
        return //base case
    split array into left_array and right_array
    → mergeSort(left_array)
    → mergeSort(right_array)

    merge(left_array, right_array, array)
}
```

Now sorted: contains left and right merged

Merge Sort Analysis

Execution time does NOT depend on initial arrangement of data

Worst Case: $O(n \log n)$ comparisons and data moves

Best Case: $O(n \log n)$ comparisons and data moves

Stable

Best we can do with comparison-based sorting that does not rely on a data structure in the worst case \Rightarrow can't beat $O(n \log n)$

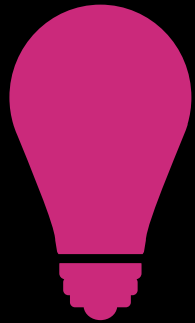
Space overhead: auxiliary array at each merge step

What we have so far



	Worst Case	Best Case
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n)$
Bubble Sort	$O(n^2)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$

Quick Sort

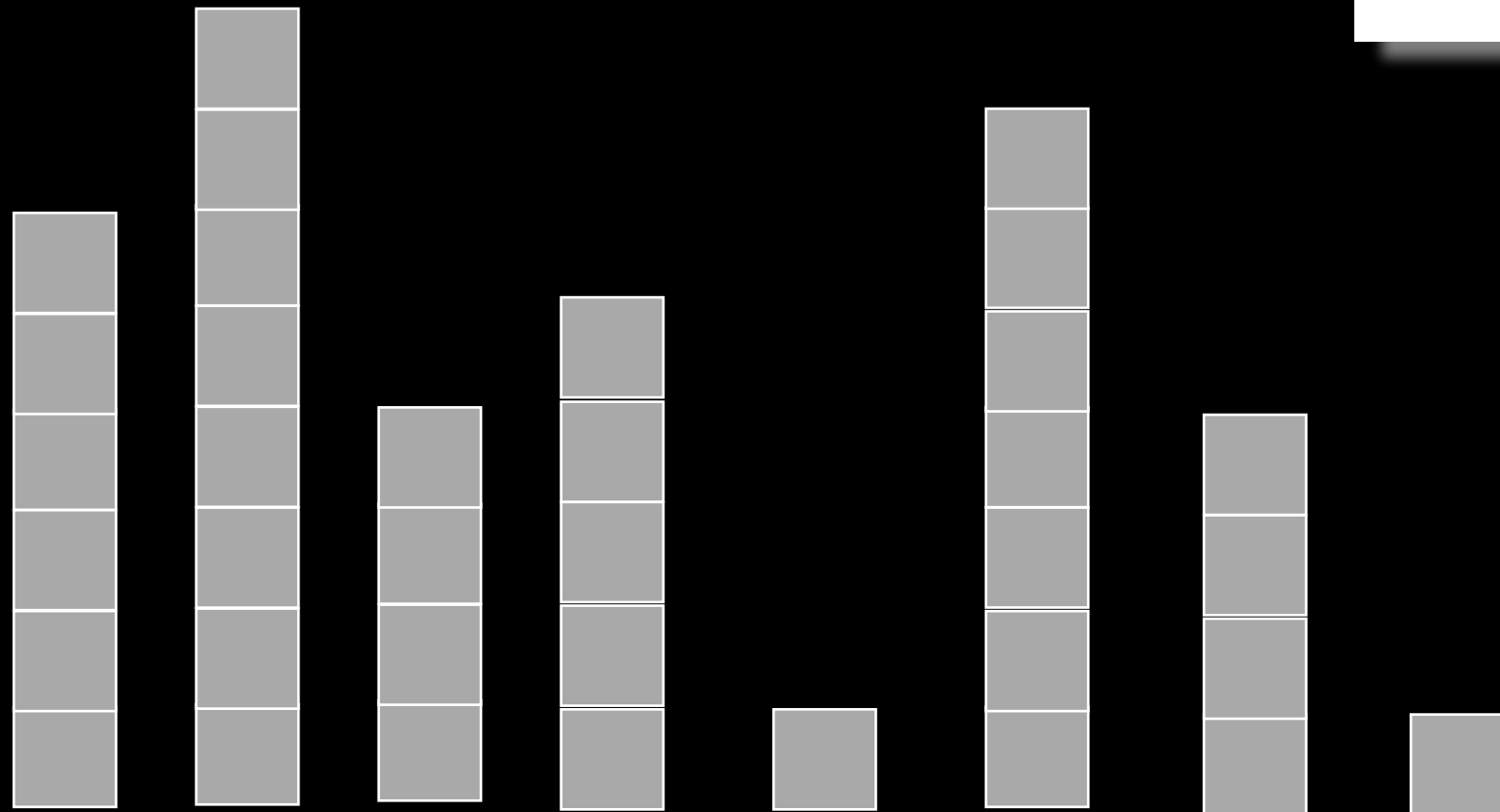
Quick Sort



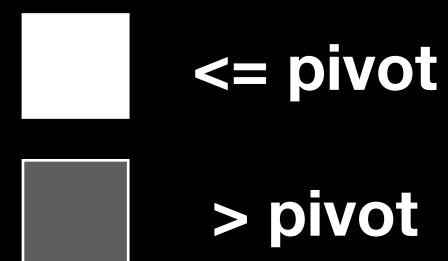
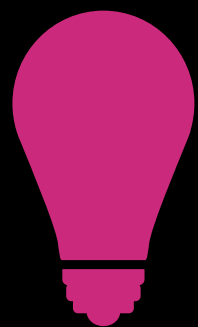
Select a pivot. Arrange other entries
s.t. entries in **left partition** are \leq pivot
and entries in **right partition** are $>$ pivot

 \leq pivot
 $>$ pivot

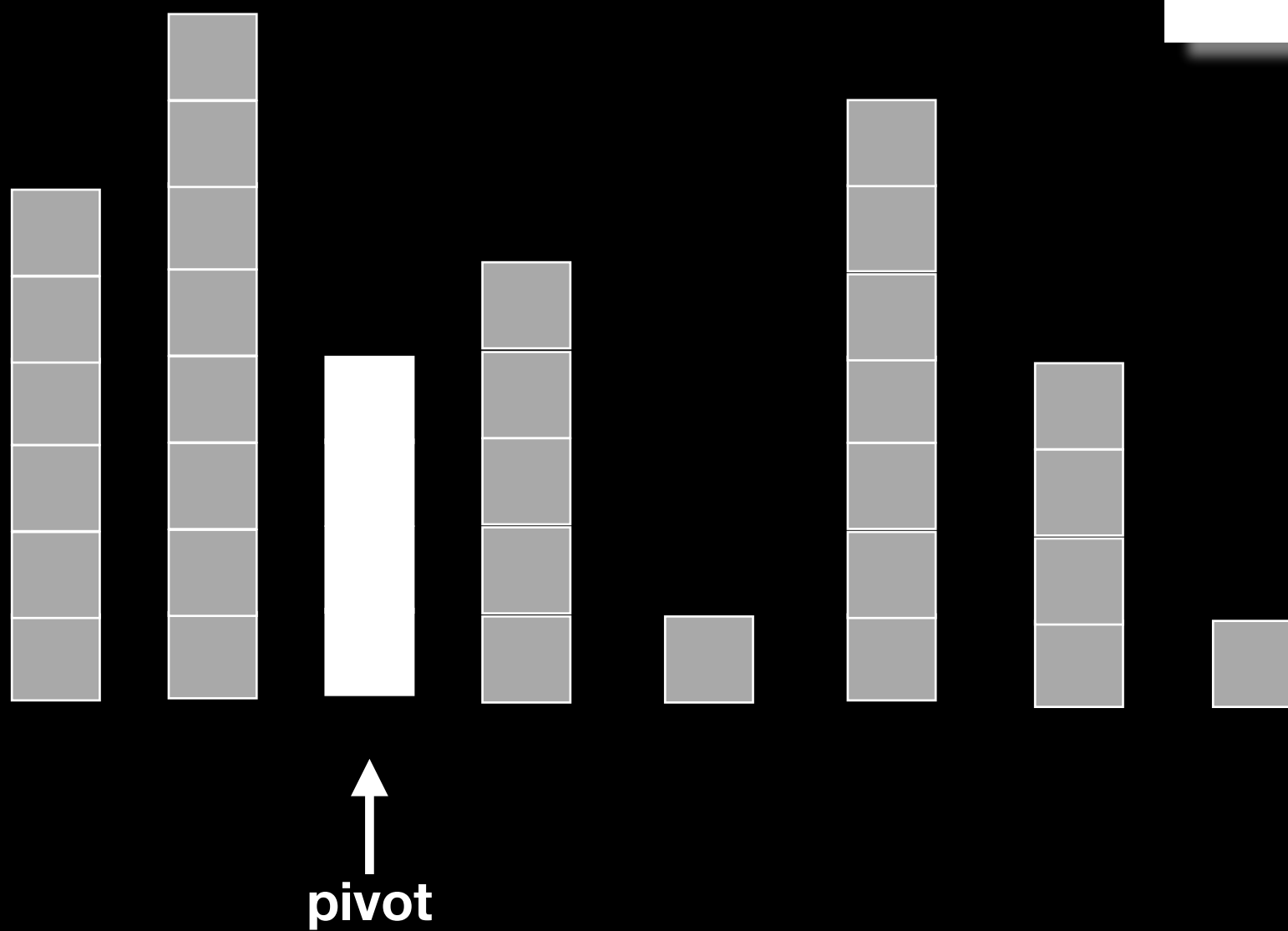
Partition



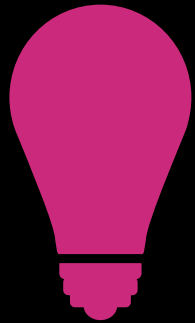
Quick Sort





Partition



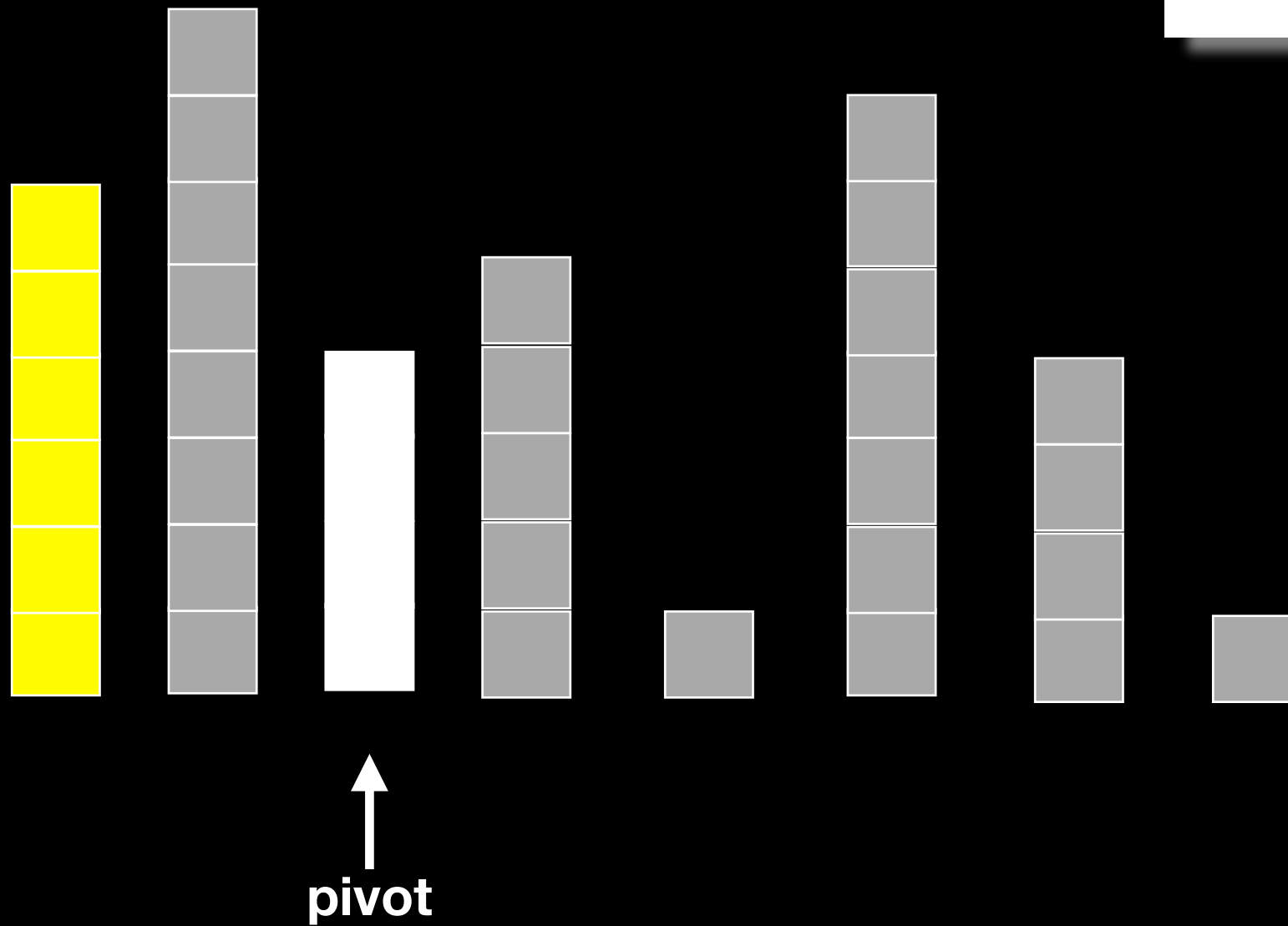
Quick Sort



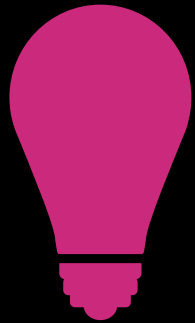
Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot

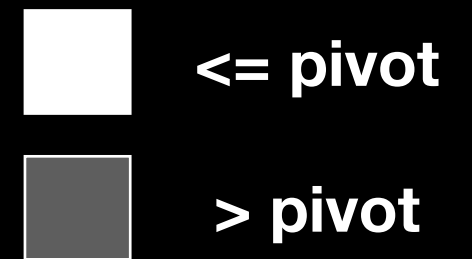
Partition



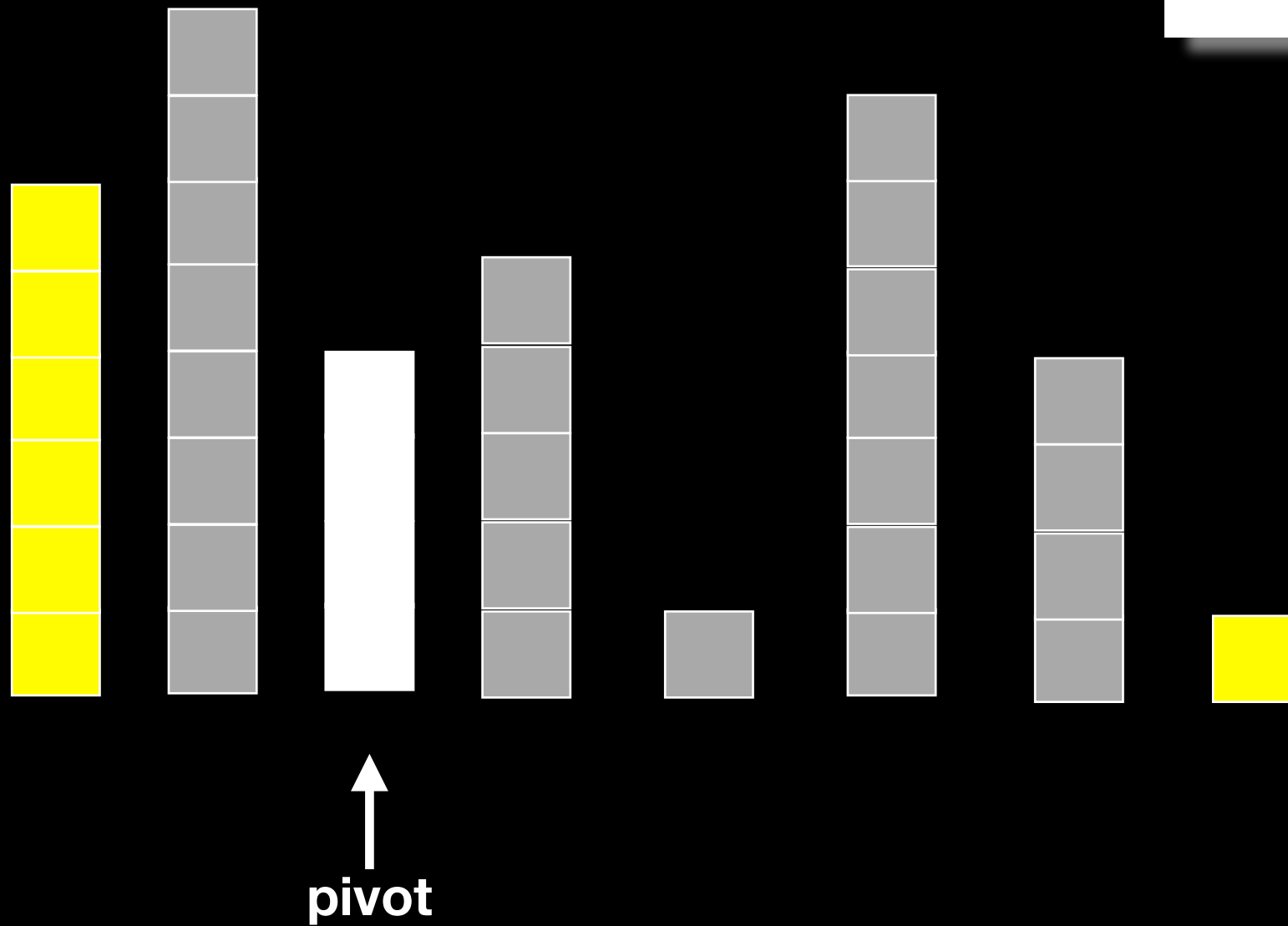
Quick Sort



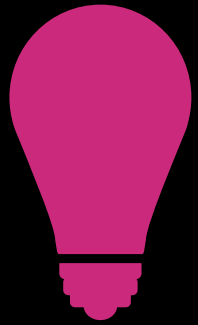
Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**





Partition



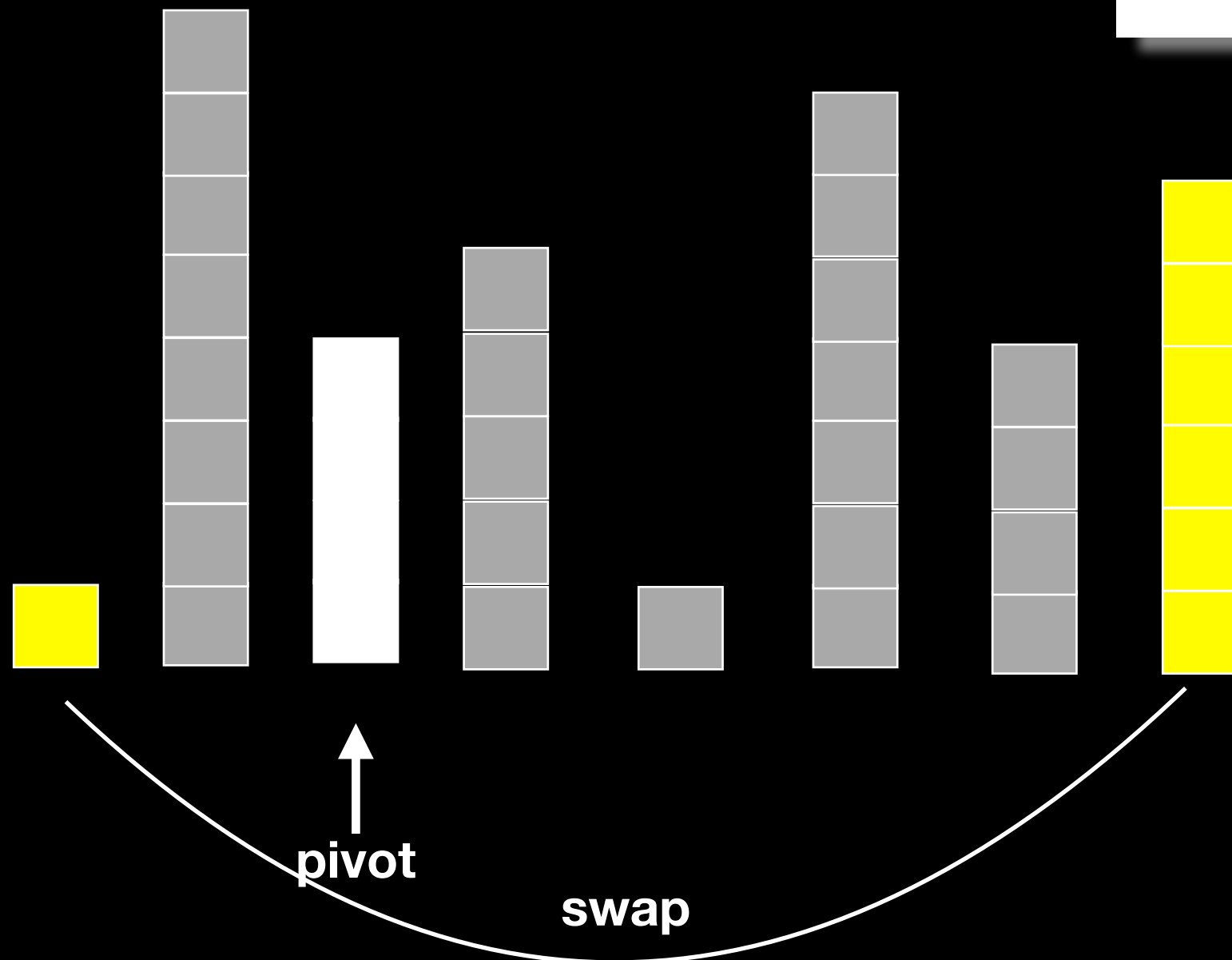
Quick Sort



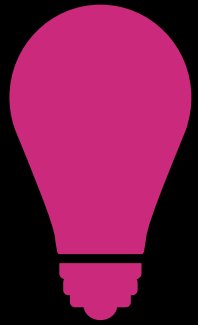
Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot



Partition



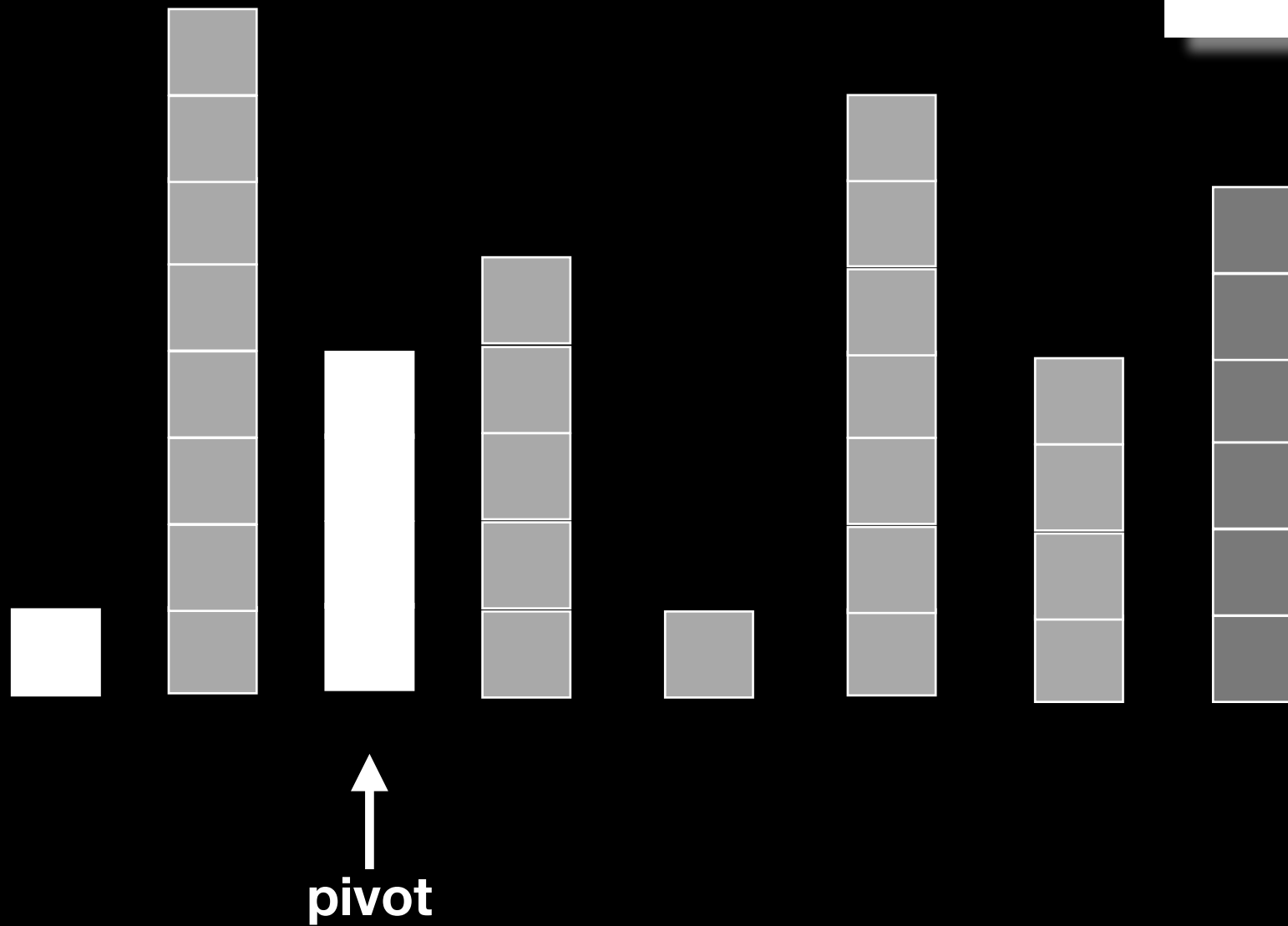
Quick Sort



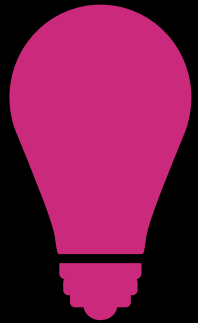
Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot



Partition



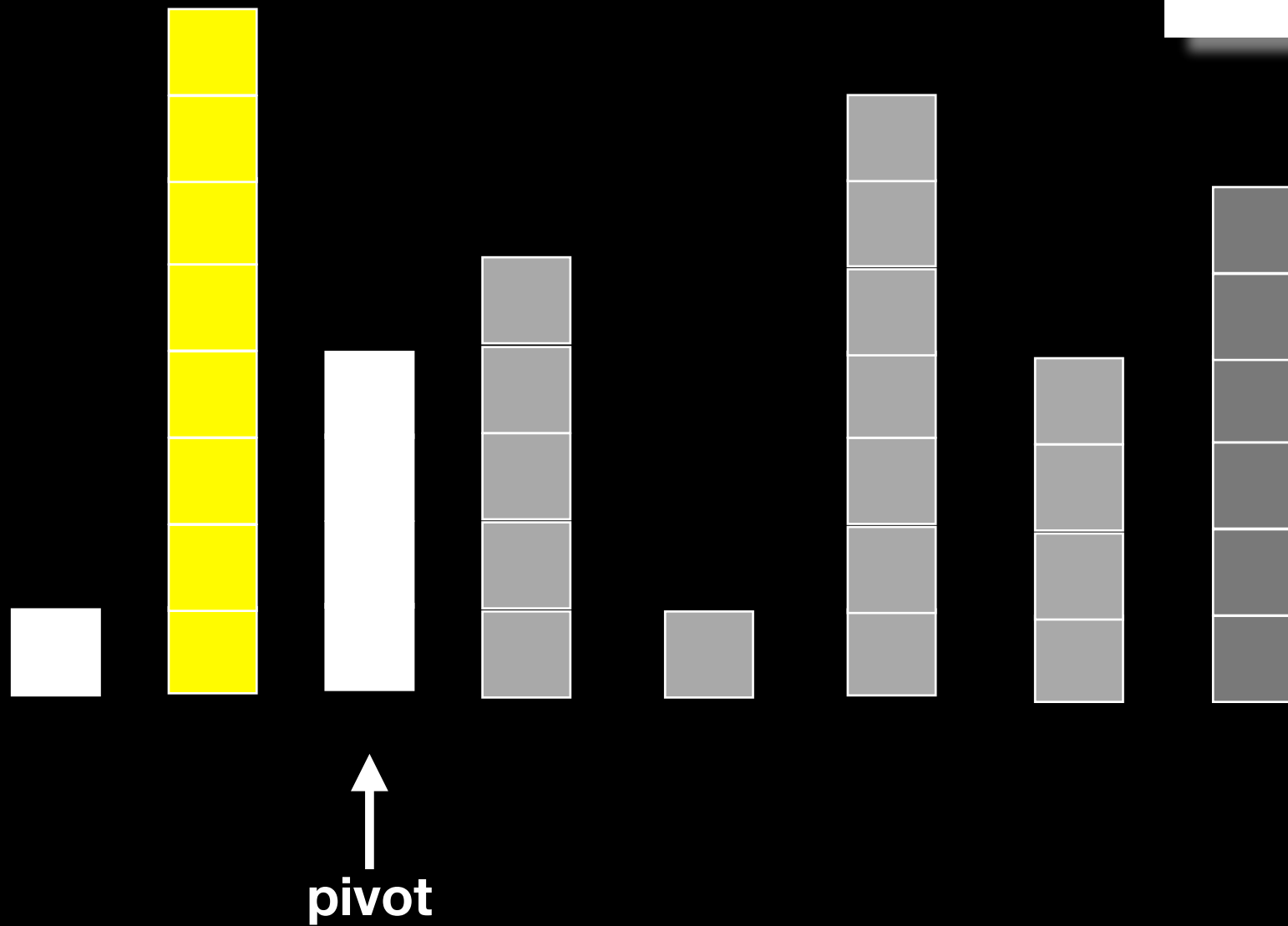
Quick Sort



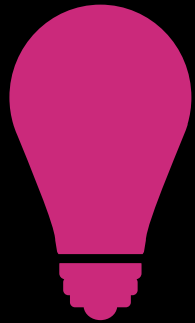
Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot



Partition



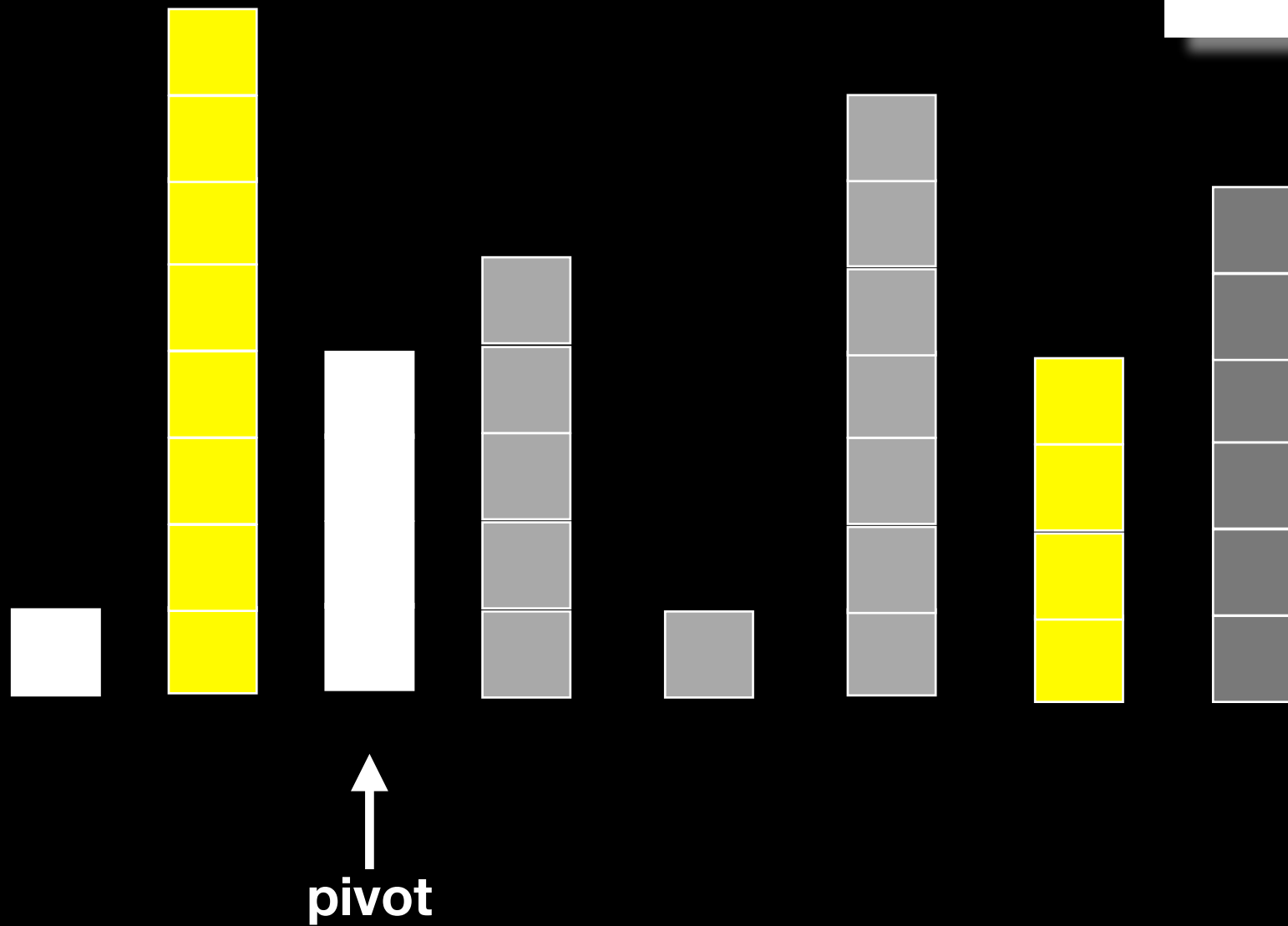
Quick Sort



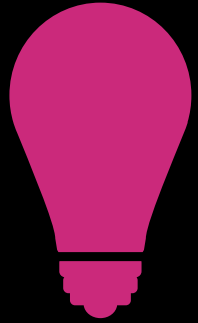
Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot



Partition



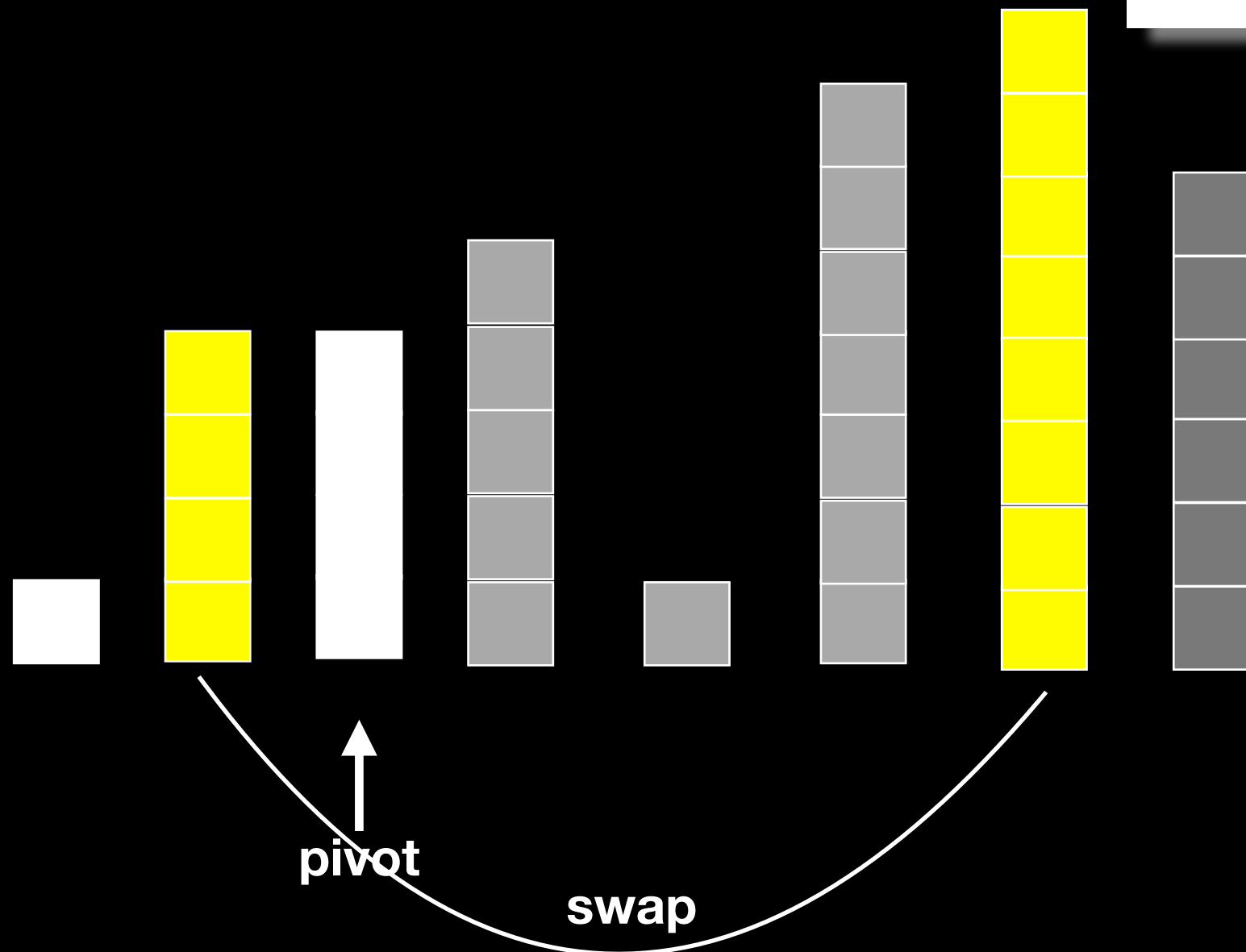
Quick Sort



Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot



Partition



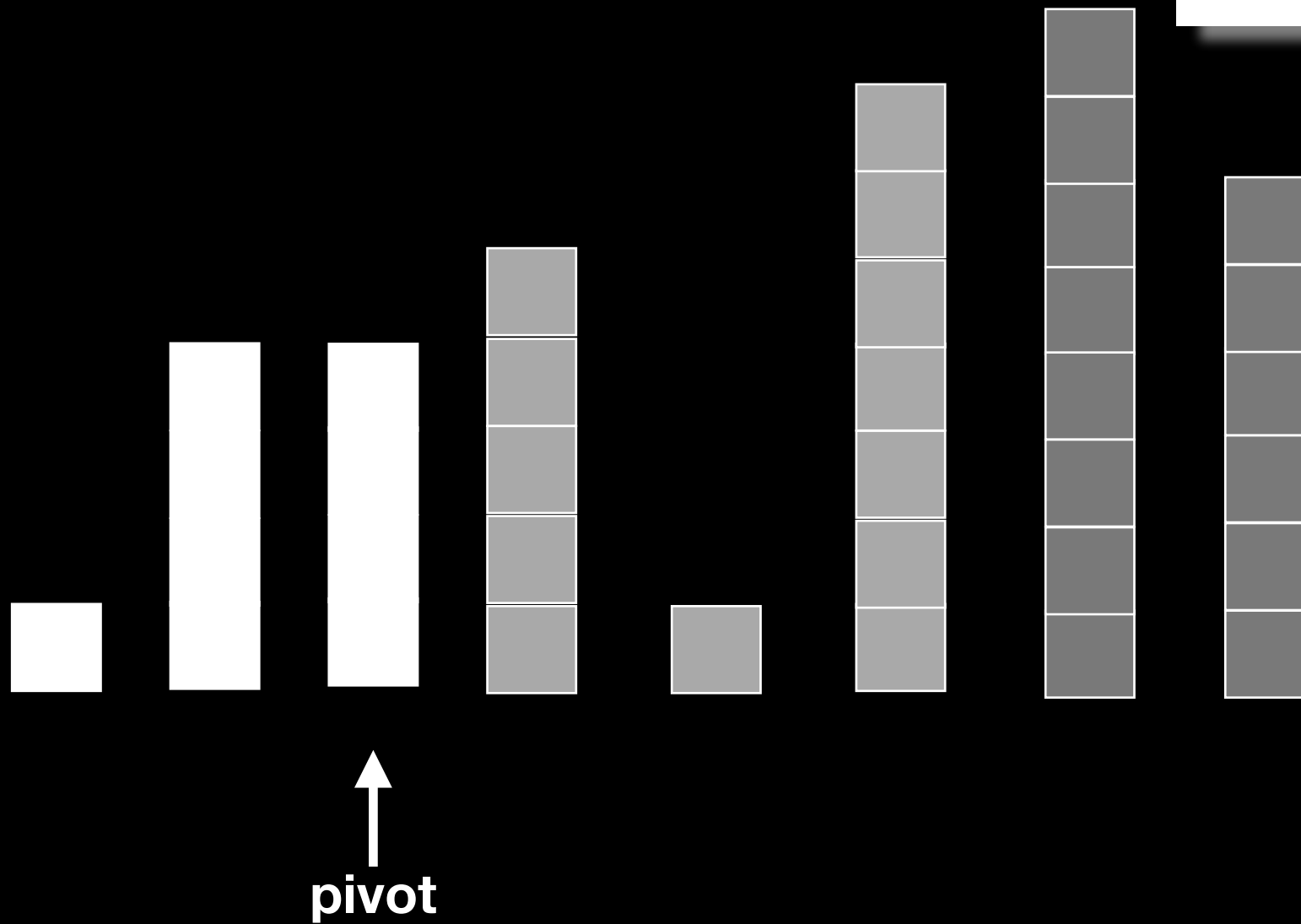
Quick Sort



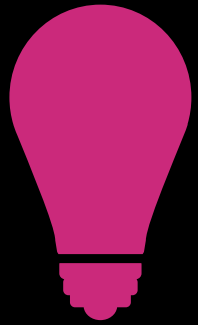
Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot



Partition



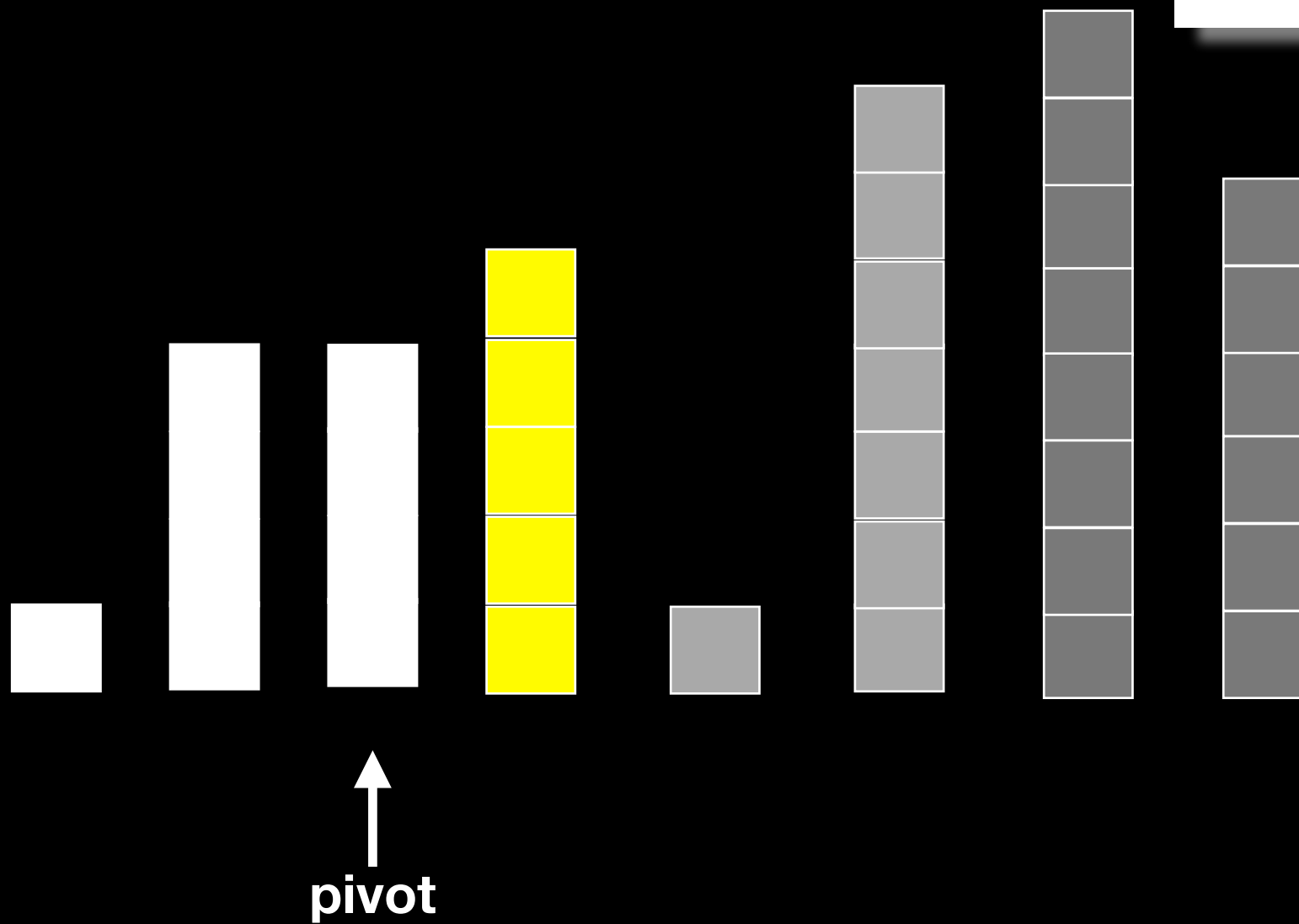
Quick Sort



Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot



Partition



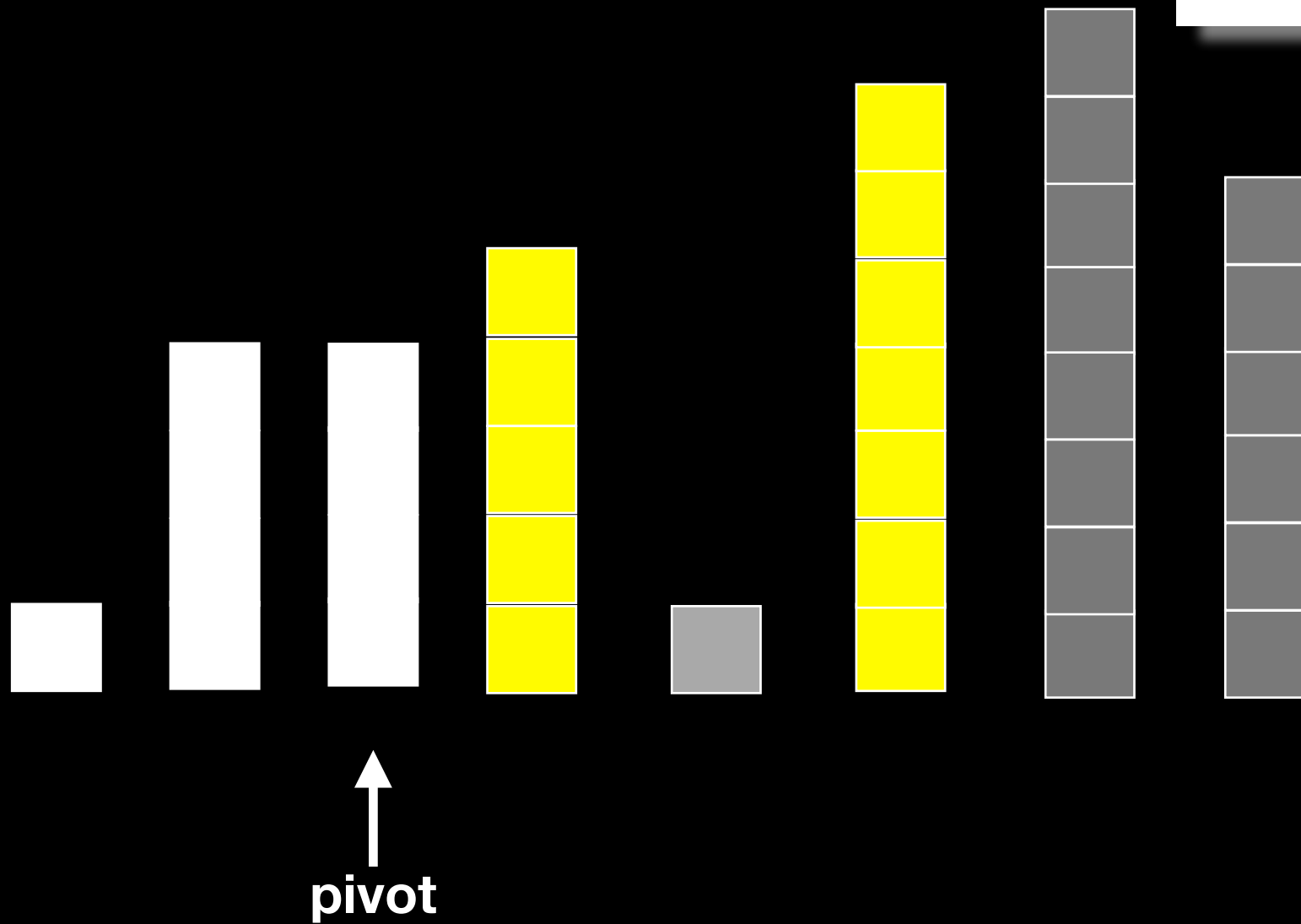
Quick Sort



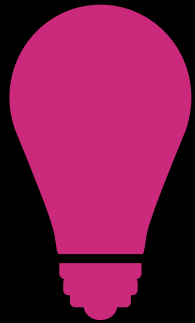
Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot



Partition



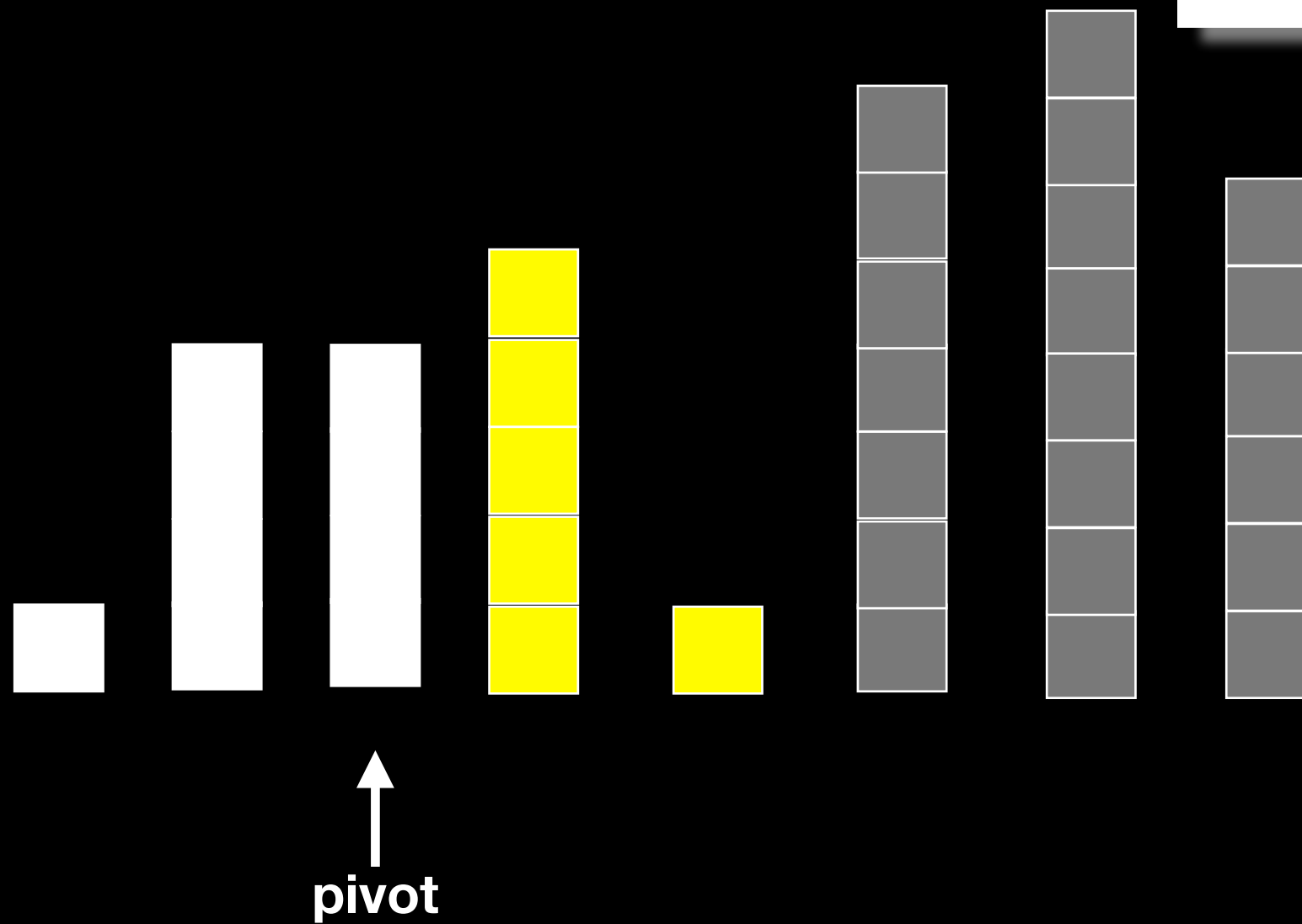
Quick Sort



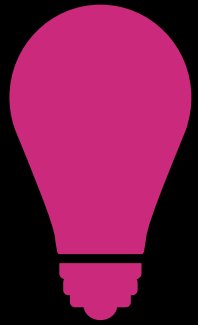
Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot

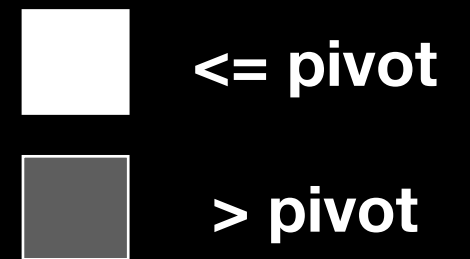
Partition



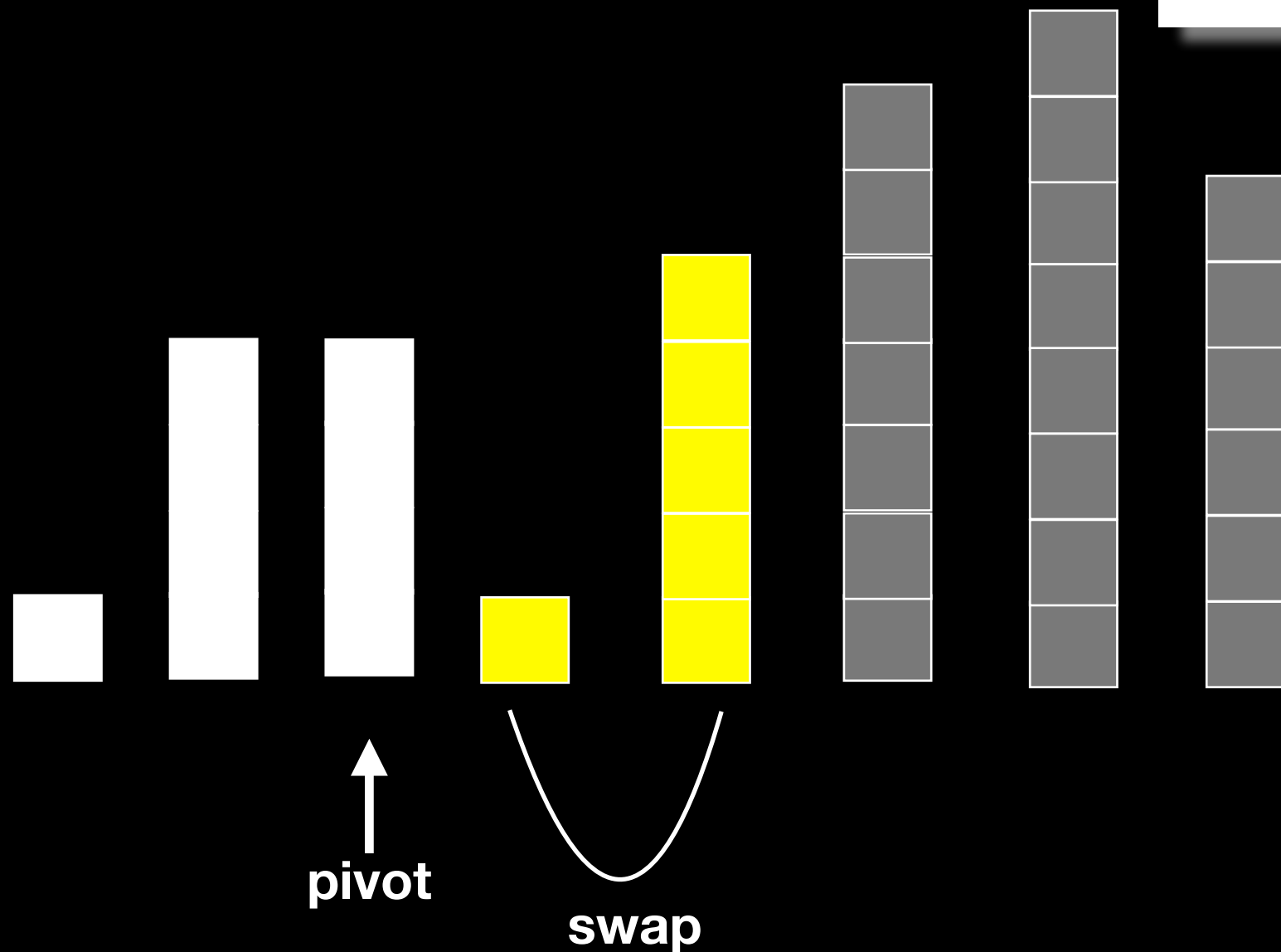
Quick Sort



Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**





Partition



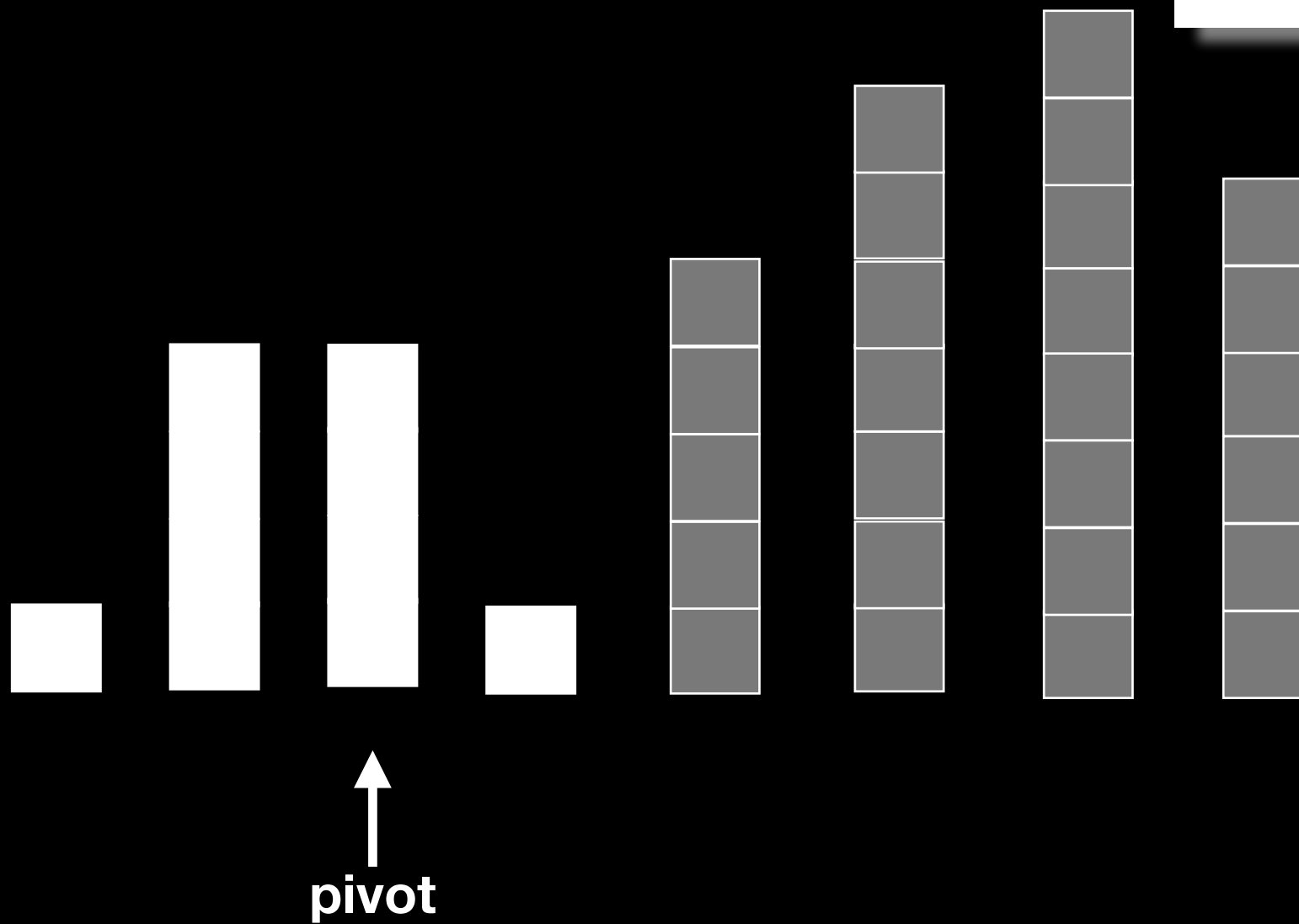
Quick Sort



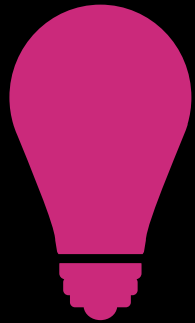
Select a **pivot**. Arrange other entries
s.t. entries in **left partition** are \leq **pivot**
and entries in **right partition** are $>$ **pivot**

 \leq pivot
 $>$ pivot

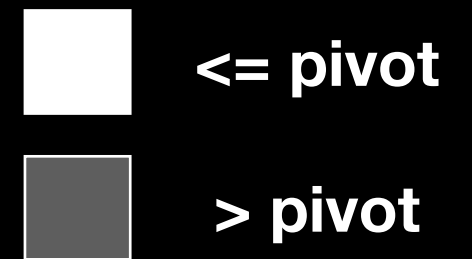
Partition



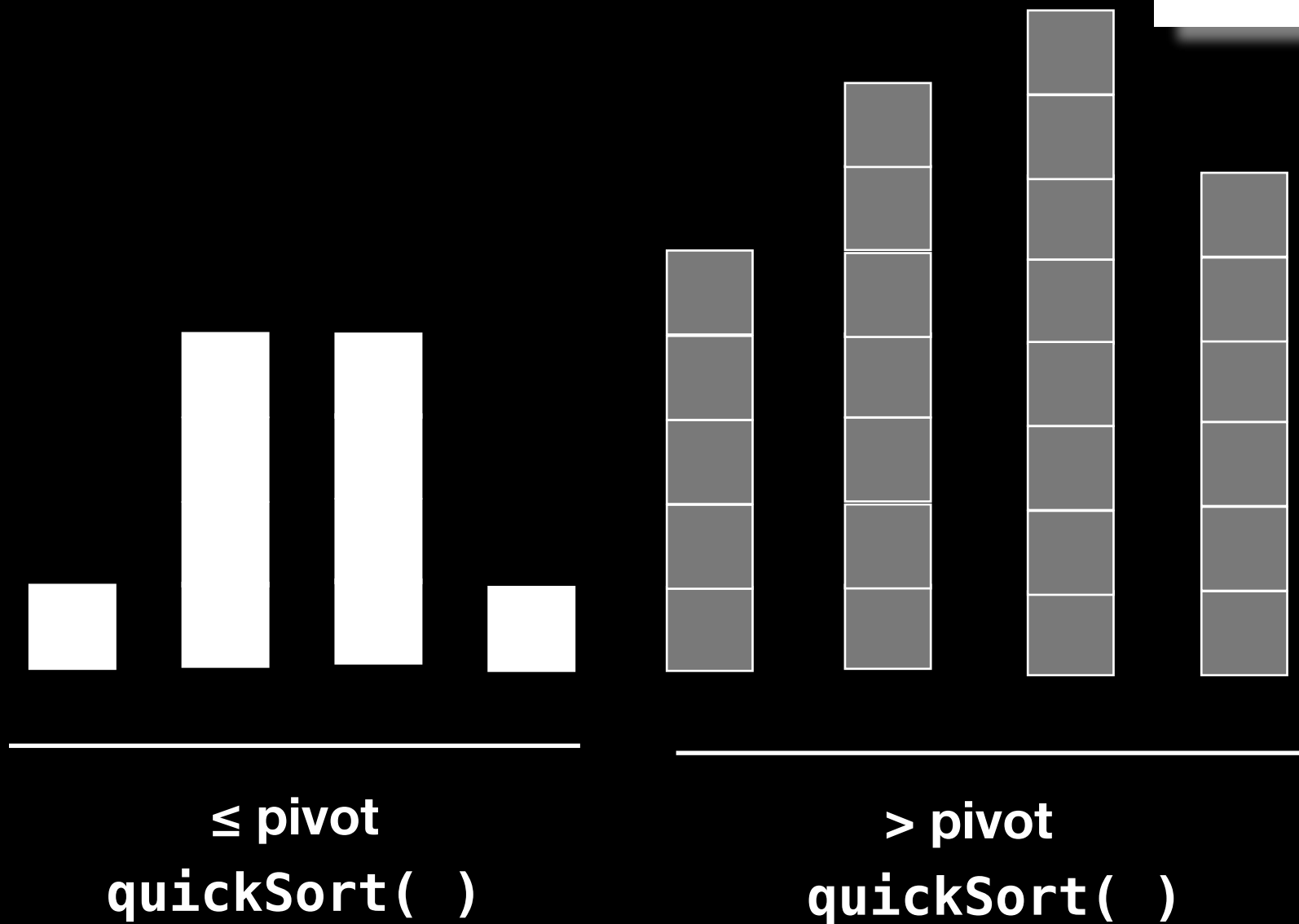
Quick Sort



Select a pivot. Arrange other entries
s.t. entries in **left partition** are \leq pivot
and entries in **right partition** are $>$ pivot



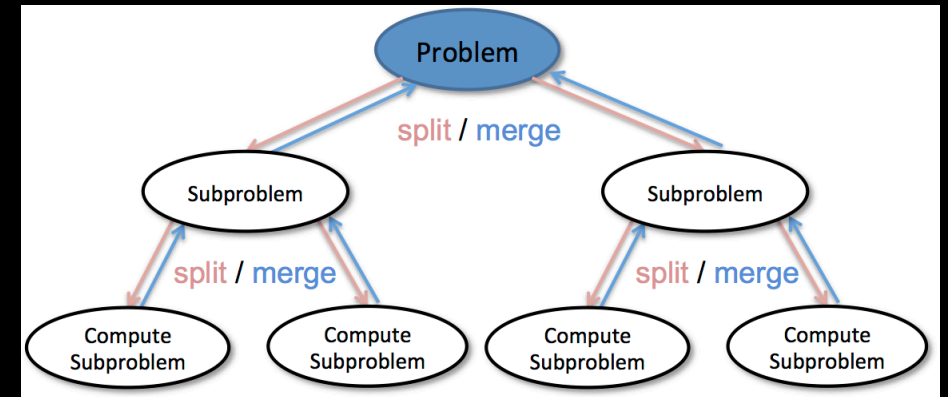
Partition



Quick Sort Analysis

Divide and Conquer

n comparisons for each partition



How many subproblems? => Depends on pivot selection

Ideally partition divides problem into two $n/2$ subproblems for $\log n$ recursive calls (Best case)

Possibly (though unlikely) each partition has 1 empty subarray for n recursive calls (Worst case)

```

template<class T>
void quickSort(T the_array[], int first, int last)
{
    if (last - first + 1 < MIN_SIZE)
    {
        insertionSort(the_array, first, last);
    }
    else
    {
        // Create the partition: S1 | Pivot | S2
        int pivot_index = partition(the_array, first, last);

        // Sort subarrays S1 and S2
        → quickSort(the_array, first, pivot_index);
        → quickSort(the_array, pivotIndex + 1, last);
    } // end if
} // end quickSort

```

Optimization

Optimization

How to select pivot?

How to select pivot?

Ideally median

Need to sort array to find median



Other ideas?

How to select pivot?

Ideally median

Need to sort array to find median



Other ideas?

Pick first



How to select pivot?

Ideally median

Need to sort array to find median



Other ideas?

Pick first, middle, last position and order them
making middle the pivot



How to select pivot?

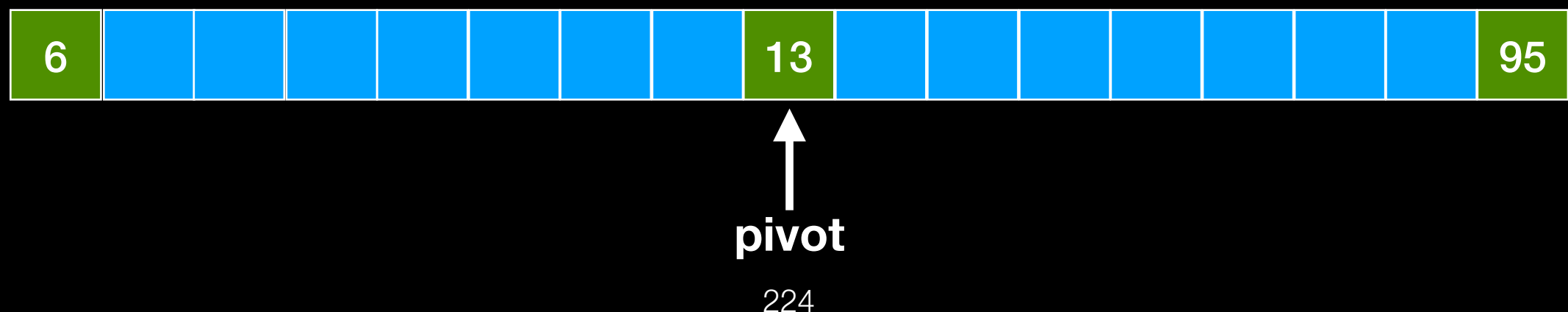
Ideally median

Need to sort array to find median



Other ideas?

Pick first, middle, last position and order them making middle the pivot



Quick Sort Analysis

Execution time DOES depend on initial arrangement of data AND on PIVOT SELECTION (luck?) => on random data can be faster than Merge Sort

Possible optimization (e.g. smart pivot selection, speed up base case, iterative instead of recursive implementation) can improve actual runtime
-> fastest comparison-based sorting algorithm **on average**

















Worst Case: $O(n^2)$ comparisons and data moves

Best Case: $O(n \log n)$ comparisons and data moves

Unstable

	Worst Case	Best Case
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n)$
Bubble Sort	$O(n^2)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$

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

 Play All	 Insertion	 Selection	 Bubble
 Random			
 Nearly Sorted			
 Reversed			

<https://www.youtube.com/watch?v=kPRA0W1kECg>

