Searching and Sorting

Tiziana Ligorio
tligorio@hunter.cuny.edu

Today's Plan

Announcements

Searching algorithms and their analysis

Sorting algorithms and their analysis

Announcements and Syllabus Check

Questions?

Searching

Looking for something!
In this discussion we will assume searching for an element in an array.

Linear search

Most intuitive

Start at first position and keep looking until you find it

```
int linearSearch(int a[], int size, int value)
{

    for (int i = 0; i < size; i++)
    {
        if (a[i] == value) {
            return i;
        }
    }
    return-1;
}
```

How long does linear search take?

If you assume value is in the array, on **average n/2**

If value is not in the array (worst case) **n**

Either way it's O(n)

What if you know **array is sorted**?
Can you do better than linear search?

In-Class Task

You are given a sorted array of integers

You can't see the values in it until you "inspect" a location

How would you search for 115? (try to do it in fewer than n steps: don't search sequentially)

You can write pseudocode or succinctly explain your algorithm

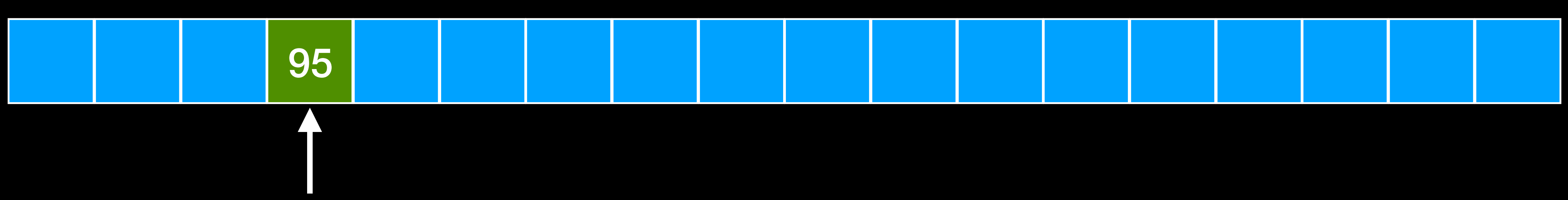
In-Class Task

You are given a sorted array of integers

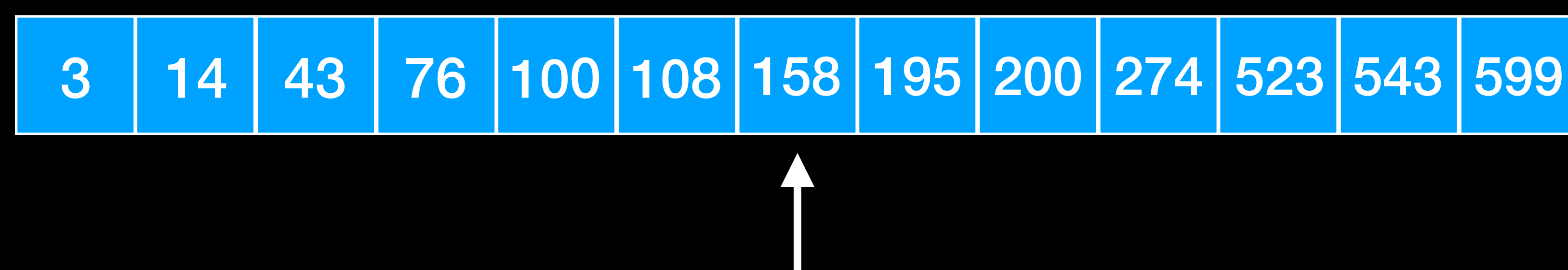
You can't see the values in it until you "inspect" a location

How would you search for 115? (try to do it in fewer than n steps: don't search sequentially)

You can write pseudocode or succinctly explain your algorithm



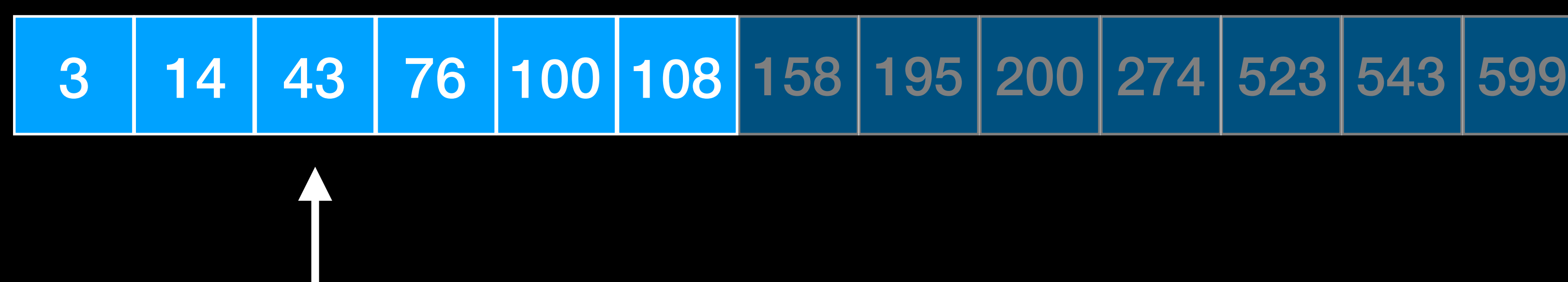
Binary Search



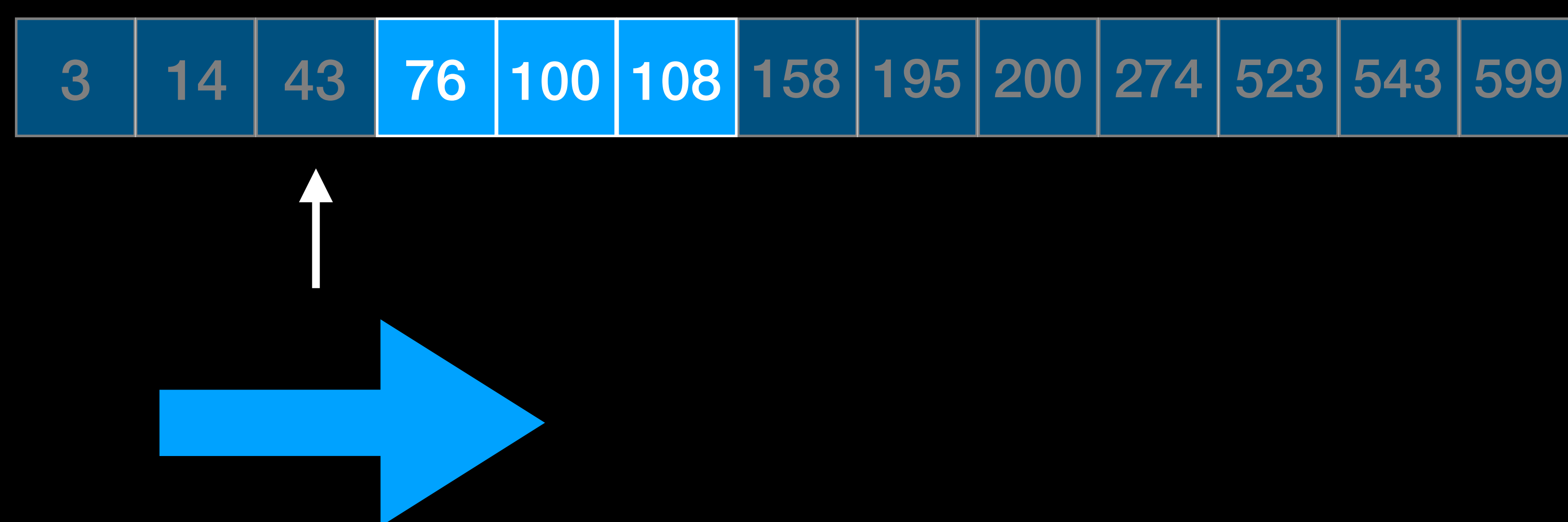
Binary Search

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

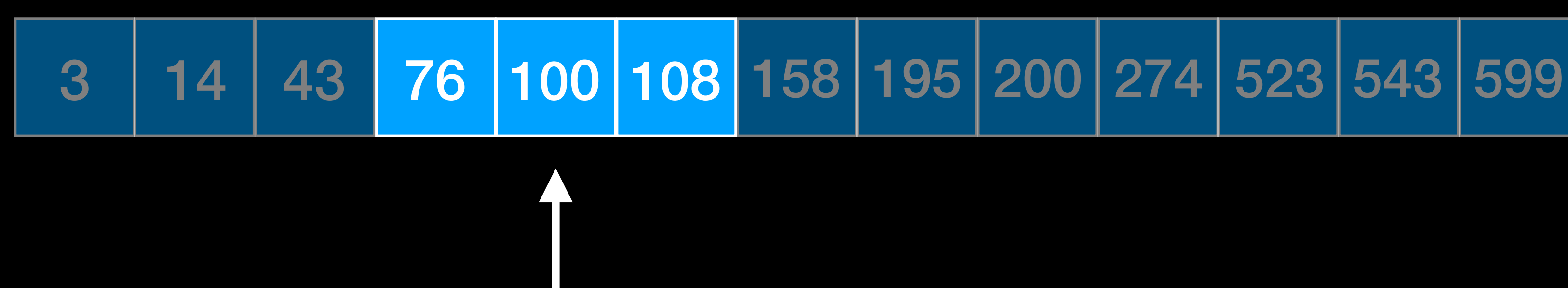
Binary Search



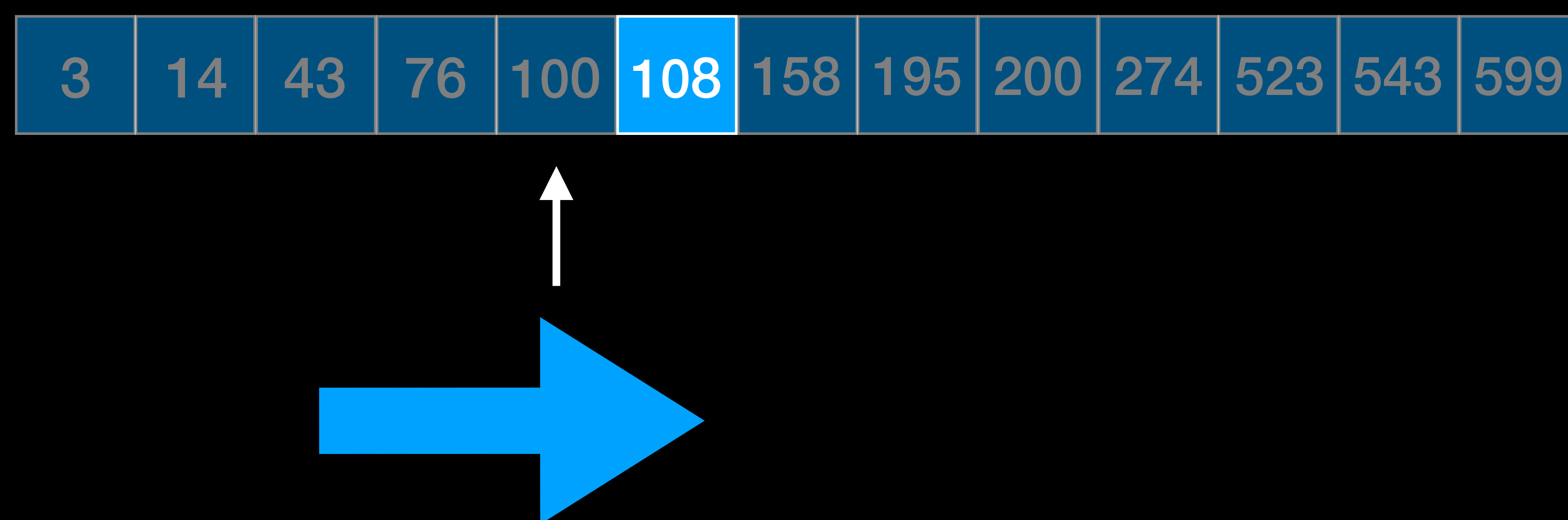
Binary Search



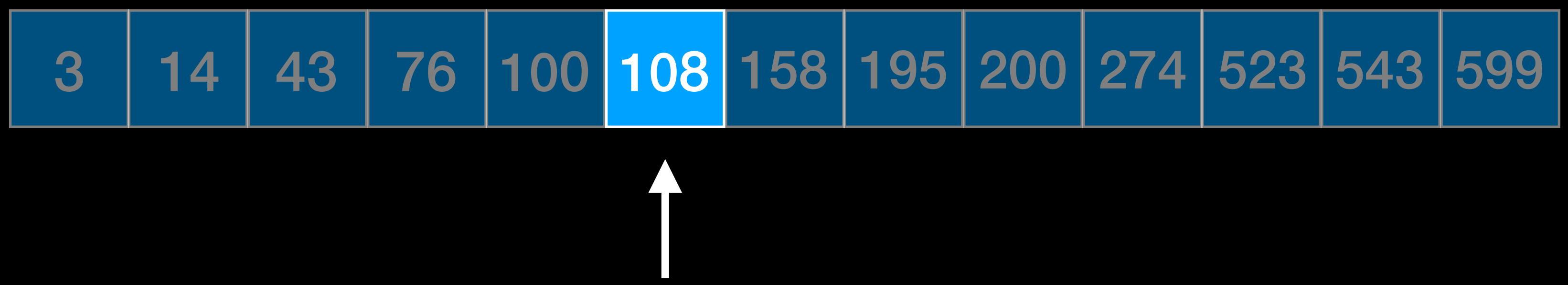
Binary Search



Binary Search



Binary Search



Binary Search

What is happening here?

Binary Search

What is happening here?

Size of search is **cut in half** at each step

Binary Search

What is happening here?

Size of search is **cut in half** at each step

The running time

Let $T(n)$ be the running time and **assume $n = 2^k$**

Simplification: assume n is a power of 2 so it can be evenly divided in two parts

$T(n) = T(n/2) + 1$

One comparison

Search lower OR upper half

Binary Search

What is happening here?

Size of search is **cut in half** at each step

Let $T(n)$ be the running time and **assume $n = 2^k$**

$T(n) = T(n/2) + 1$

$T(n/2) = T(n/4) + 1$

One comparison

Search lower OR upper half of n/2

Binary Search

What is happening here?

Size of search is **cut in half** at each step

Let $T(n)$ be the running time and **assume $n = 2^k$**

$T(n) = \boxed{T(n/2)} + 1$

$T(n/2) = T(n/4) + 1$

$T(n) = \boxed{T(n/4) + 1} + 1$

Binary Search

What is happening here?

Size of search is **cut in half** at each step

Let $T(n)$ be the running time and **assume $n = 2^k$**

$$T(n) = T(n/\underbrace{2}_{2^1}) + \underbrace{1}_{1}$$

$$T(n) = T(n/\underbrace{4}_{2^2}) + \underbrace{2}_{2}$$

. . .

Binary Search

What is happening here?

Size of search is **cut in half** at each step

Let $T(n)$ be the running time and **assume $n = 2^k$**

$T(n) = T(n/2) + 1$

$T(n) = T(n/4) + 2$

. . .

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

Binary Search

What is happening here?

Size of search is **cut in half** at each step

Let $T(n)$ be the running time and **assume $n = 2^k$**

$T(n) = T(n/2) + 1$

$T(n) = T(n/4) + 2$

. . .

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

$T(n) = T(1) + \log_2(n)$

The number to which I need to raise 2 to get n
And we said $n = 2^k$

$n/n = 1$

Binary Search

What is happening here?

Size of search is **cut in half** at each step

Let $T(n)$ be the running time and **assume $n = 2^k$**

$T(n) = T(n/2) + 1$

$T(n) = T(n/4) + 2$

. . .

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

$T(n) = T(1) + \log_2(n)$

Binary search is $O(\log(n))$

Sorting

Rearranging a sequence into sorted order!

Several approaches

Can do it in may ways

What is the best way?

Let's find out using Big-O

In-Class Task

Assuming you do not have a global view of the array but can see one position at a time, how would you sort this? Write pseudocode or succinctly explain

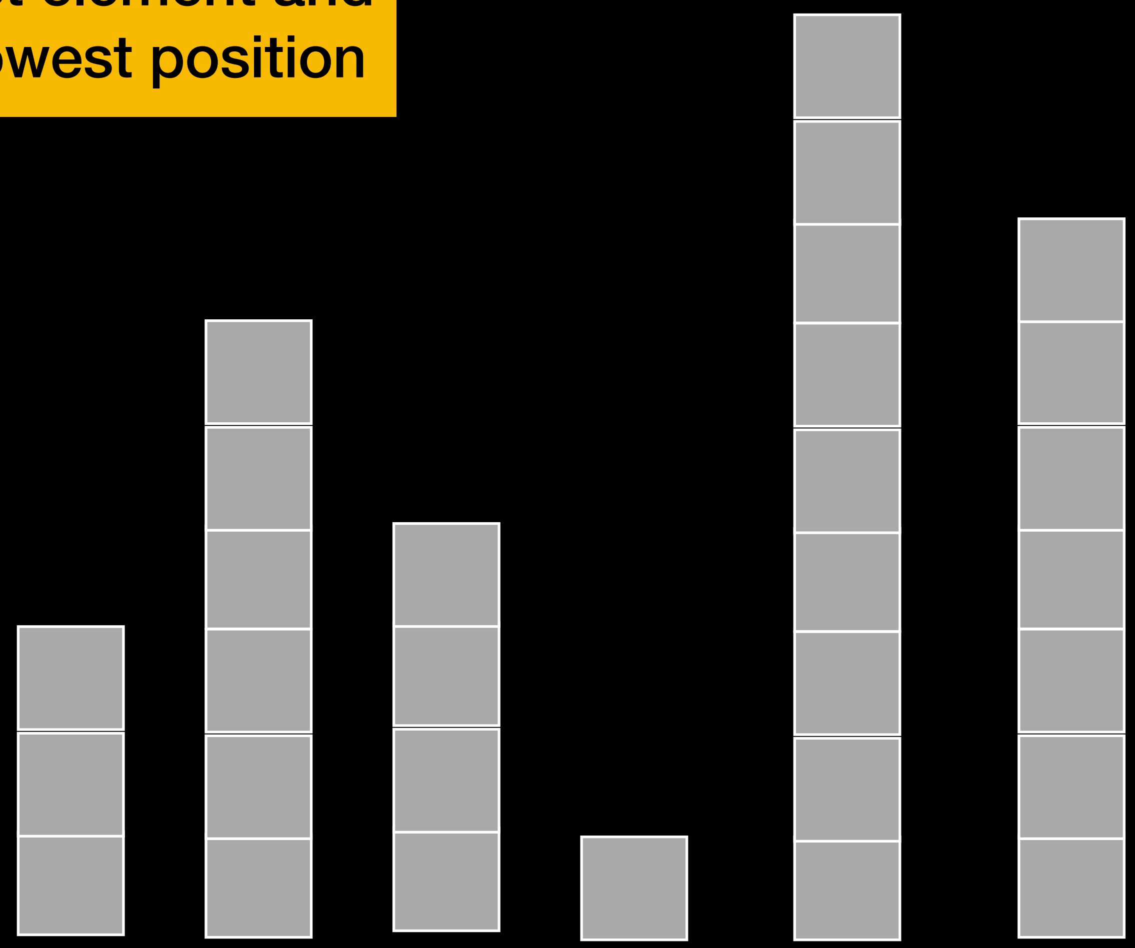
543	3	523	76	200	158	195	108	43	274	100	14	599

Selection Sort

Unsorted

Sorted

Find smallest element and move it at lowest position

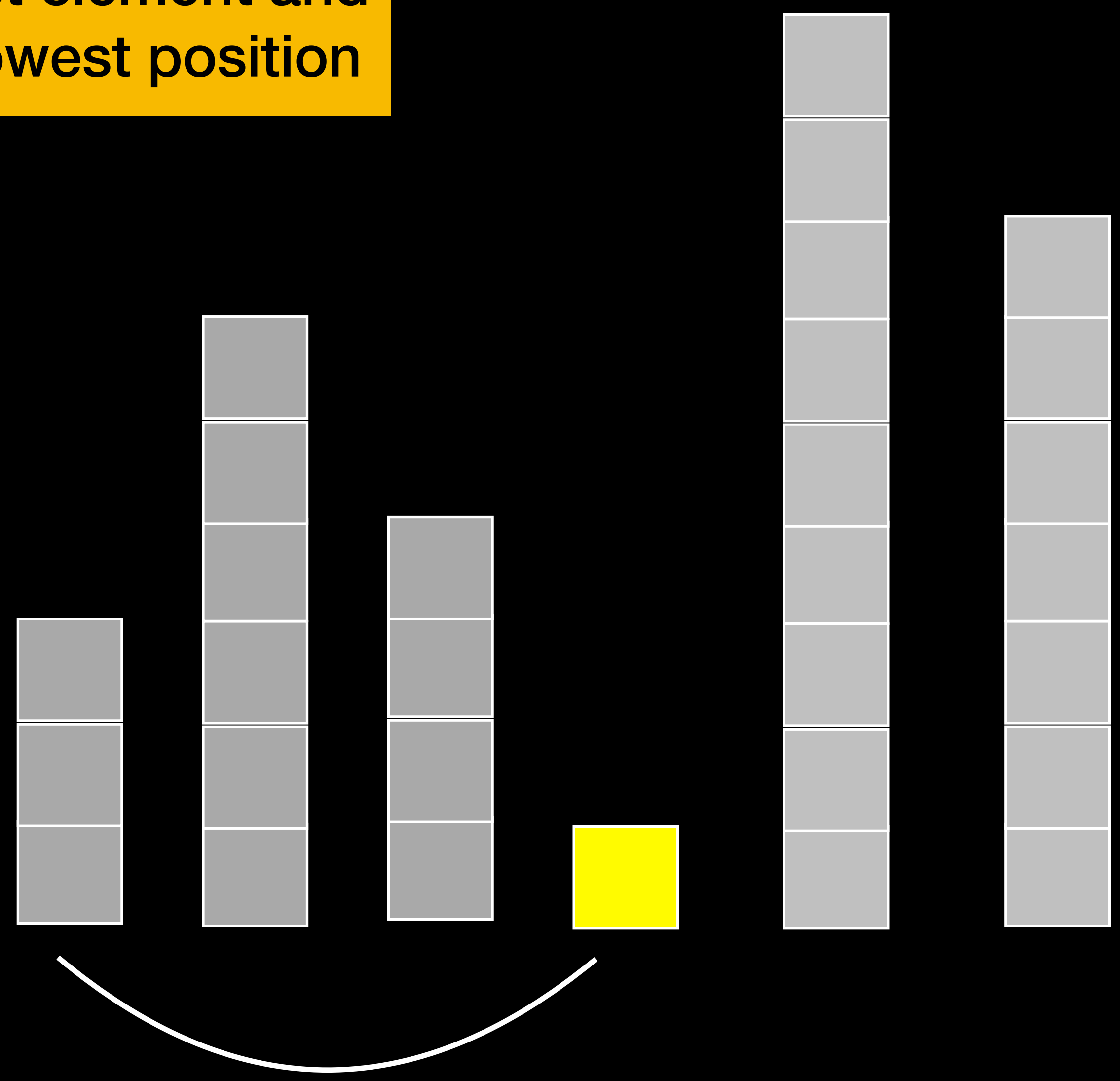


Selection Sort

Unsorted

Sorted

Find smallest element and move it at lowest position

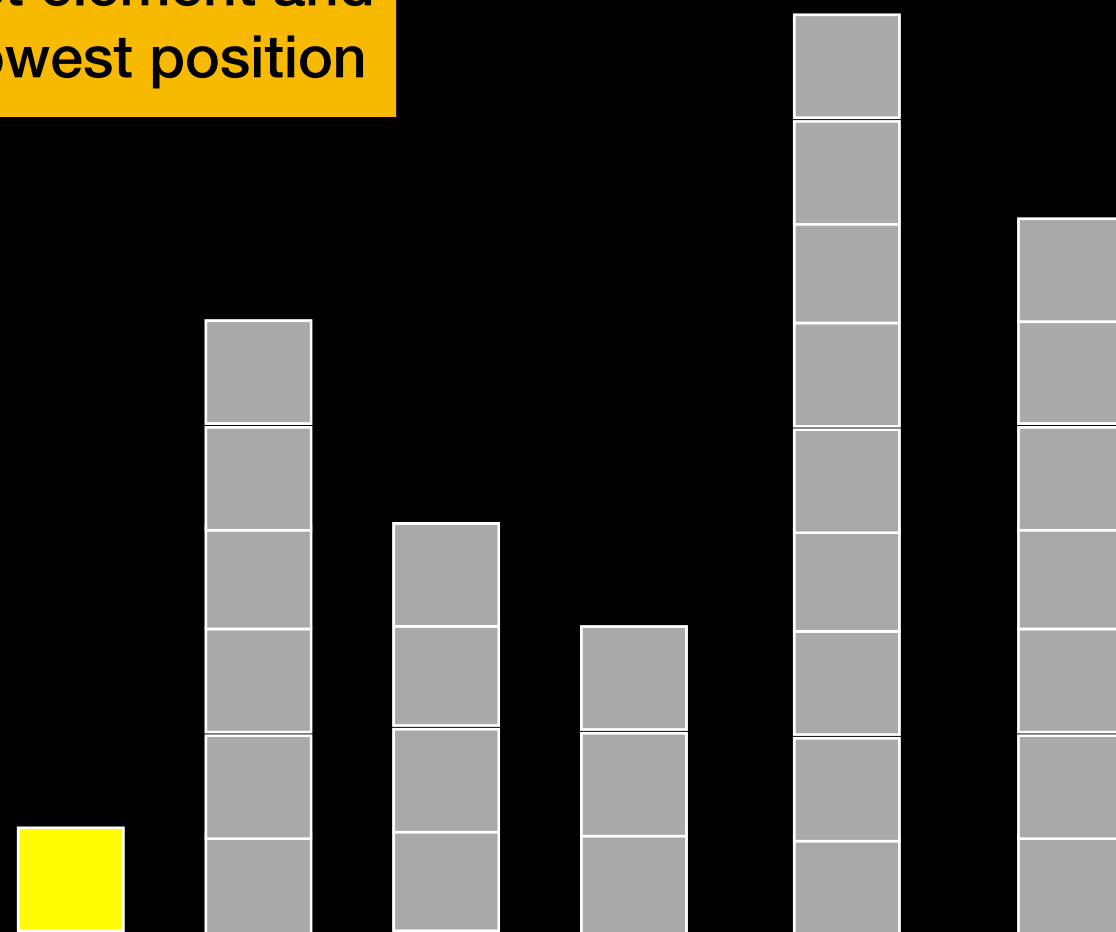


Selection Sort

Unsorted

Sorted

Find smallest element and move it at lowest position

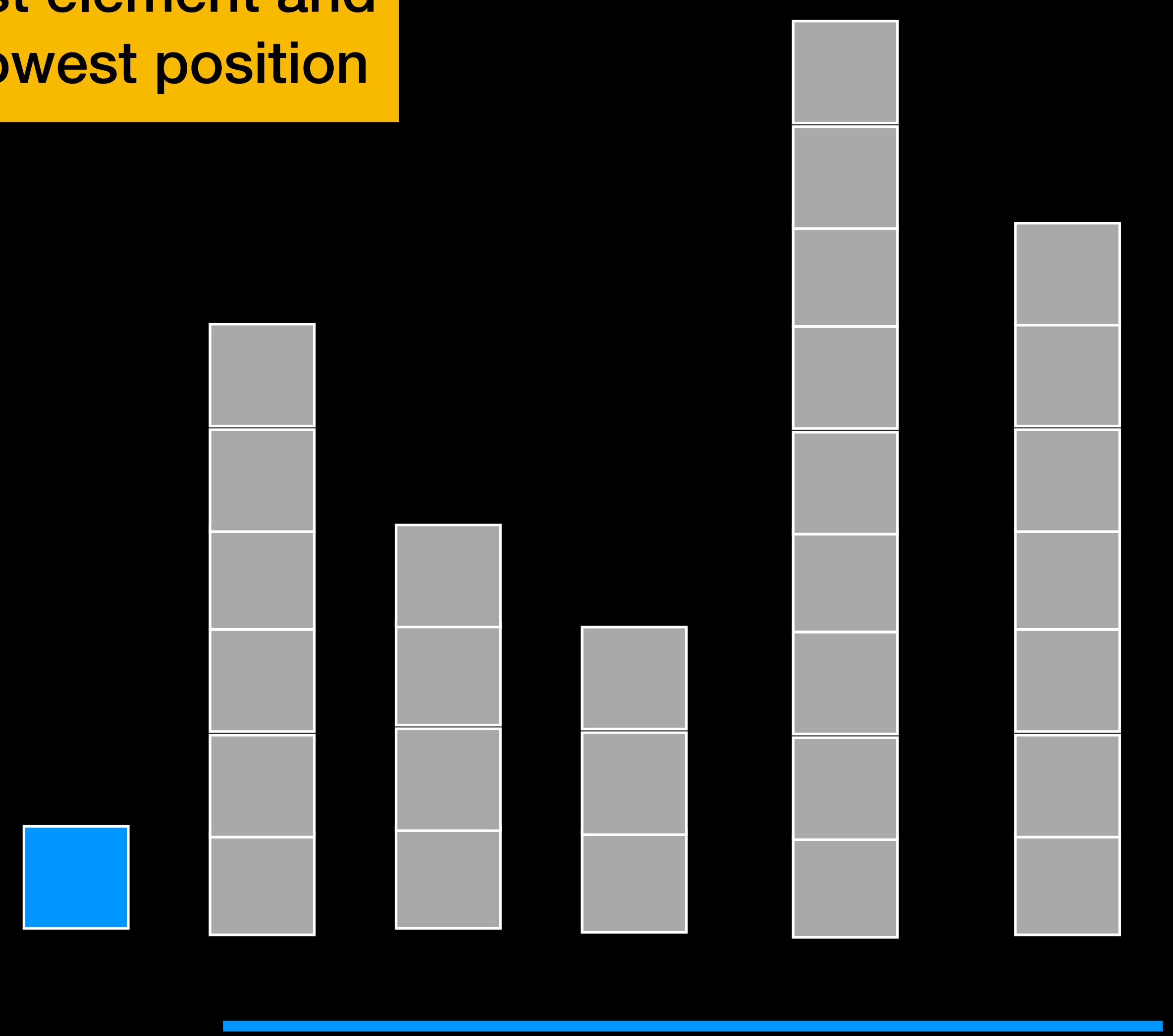


Selection Sort

Unsorted

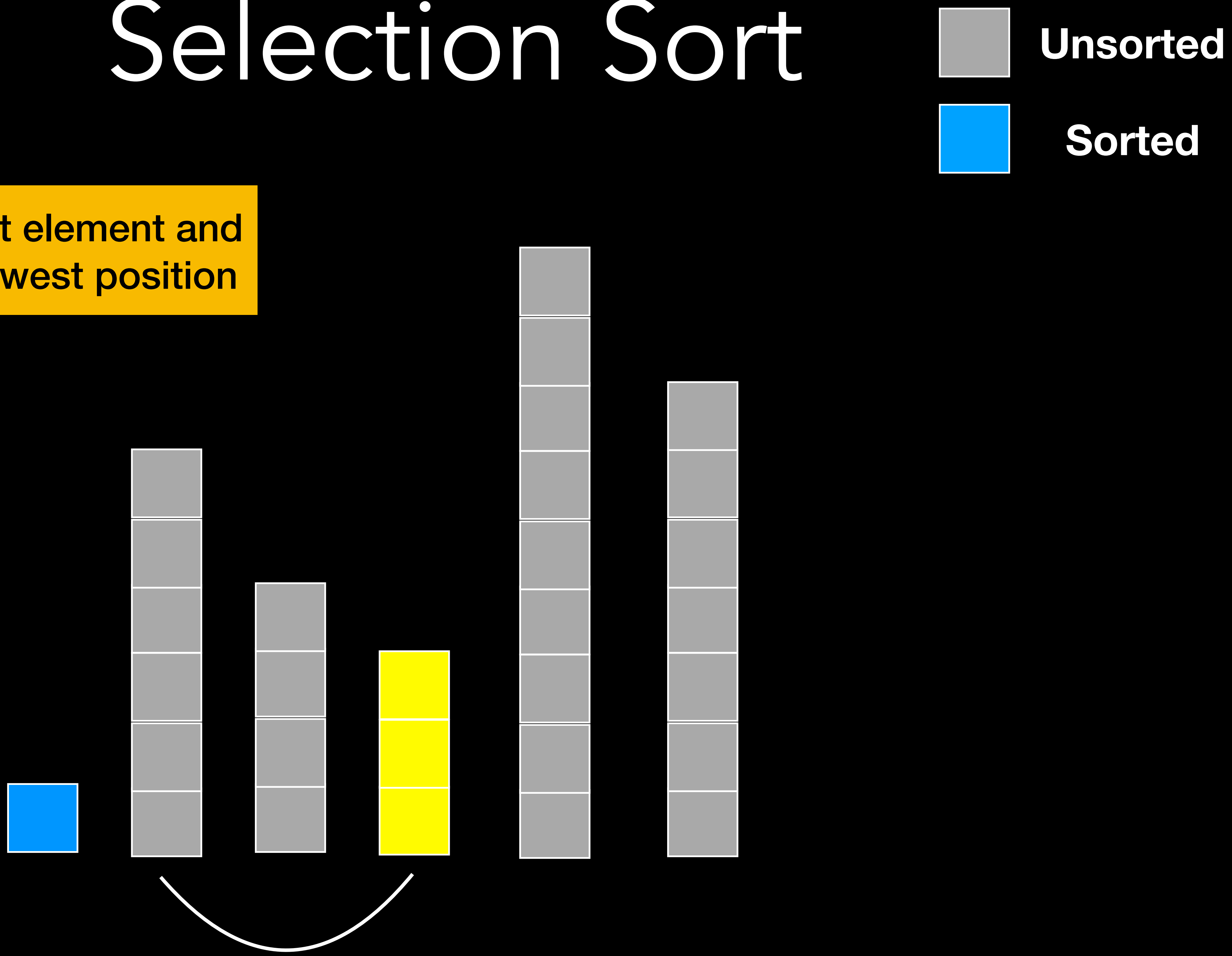
Sorted

Find smallest element and move it at lowest position



Selection Sort

Find smallest element and move it at lowest position

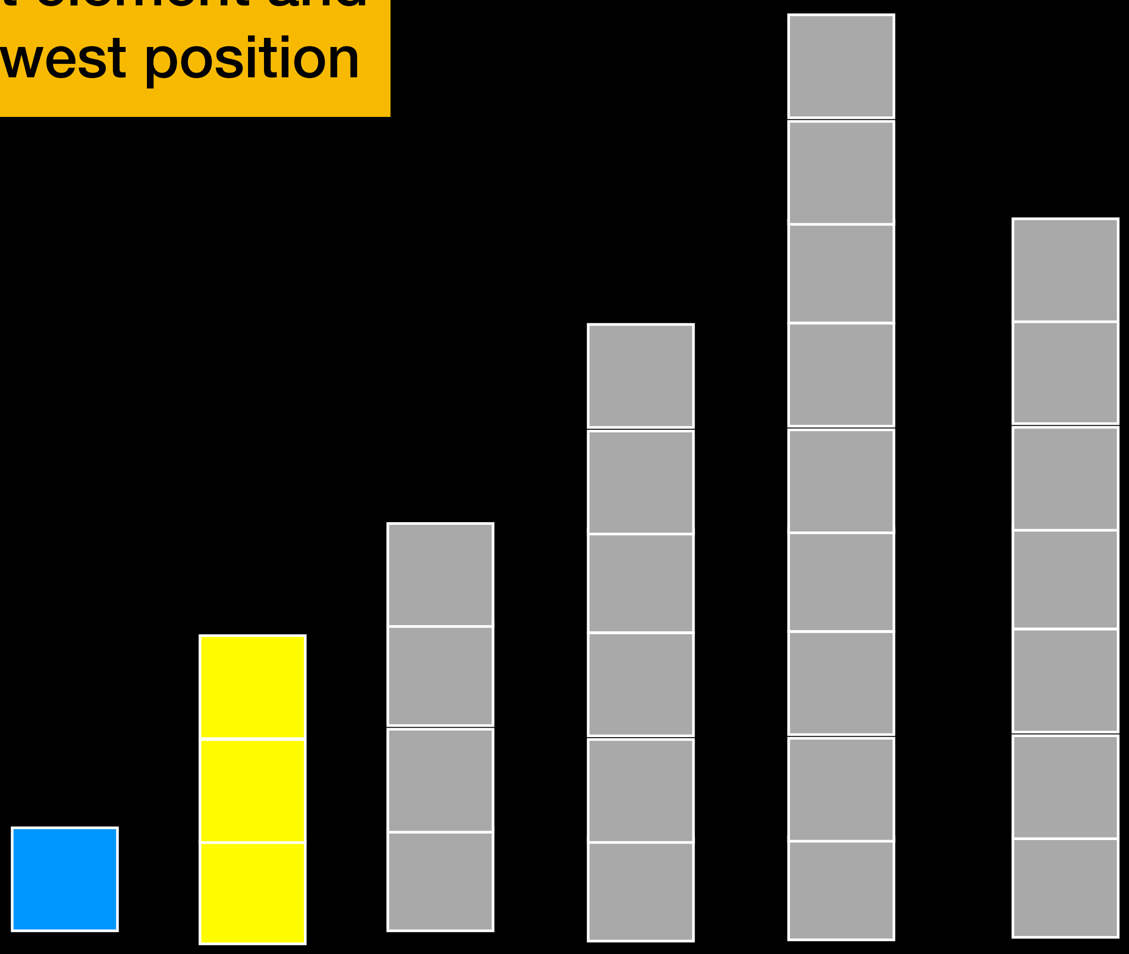


Selection Sort

Unsorted

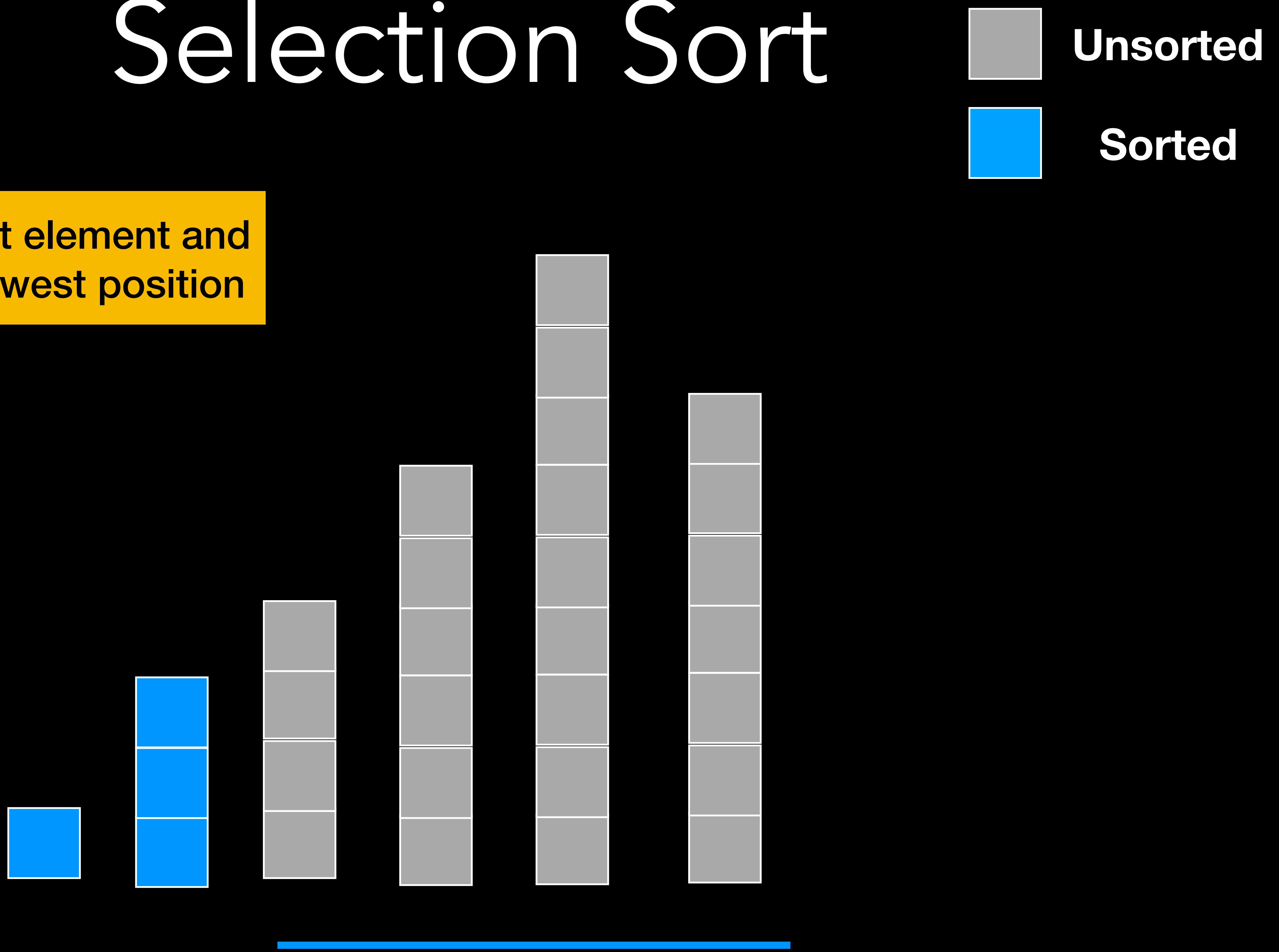
Sorted

Find smallest element and move it at lowest position



Selection Sort

Find smallest element and move it at lowest position

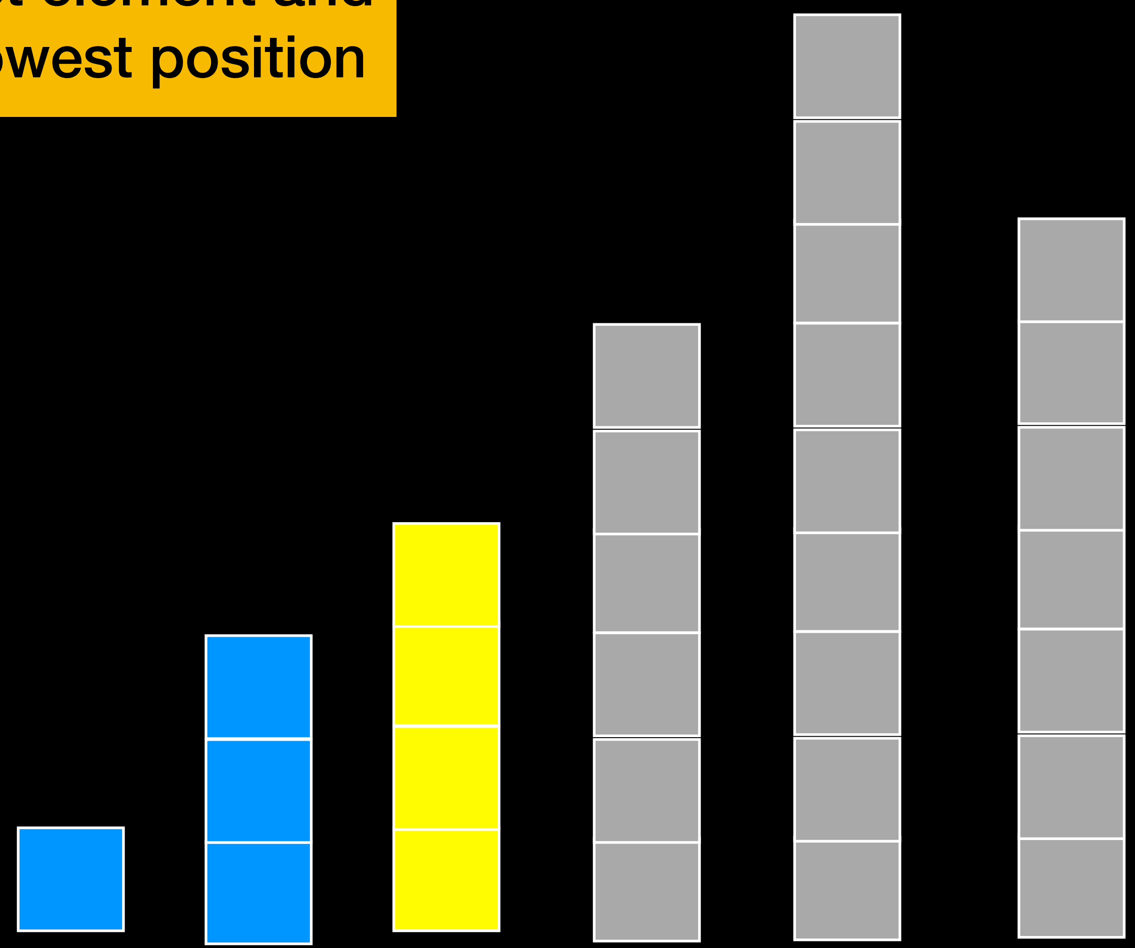


Selection Sort

Unsorted

Sorted

Find smallest element and move it at lowest position

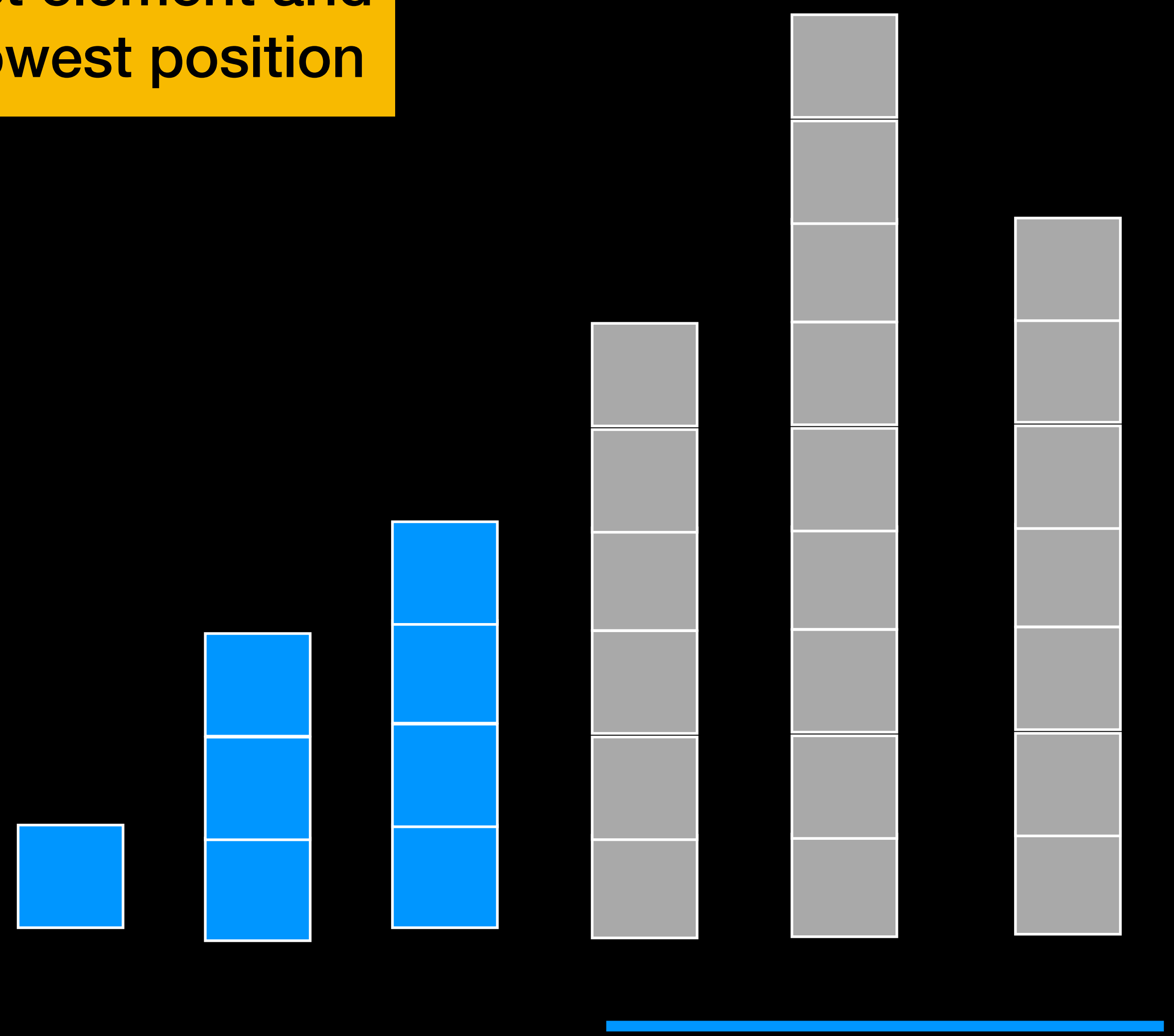


Selection Sort

Unsorted

Sorted

Find smallest element and move it at lowest position

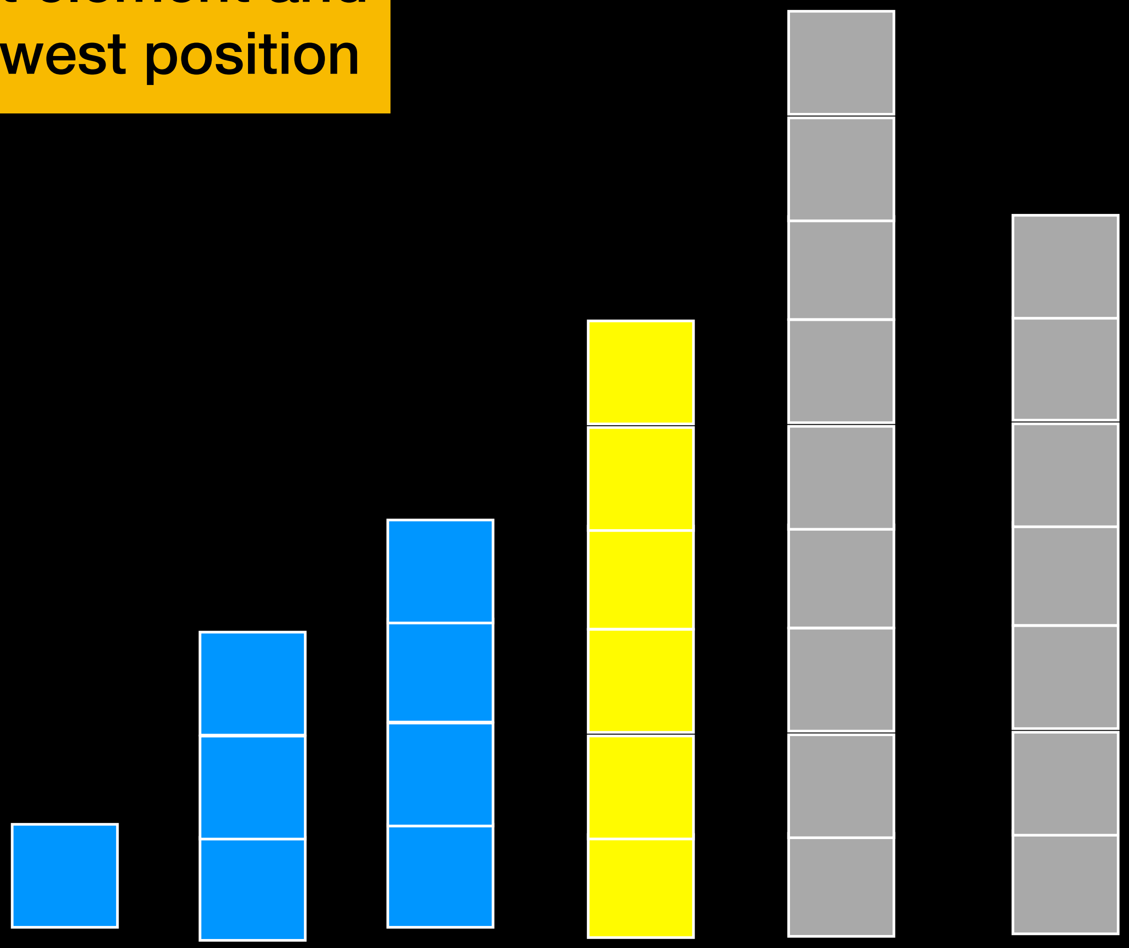


Selection Sort

Unsorted

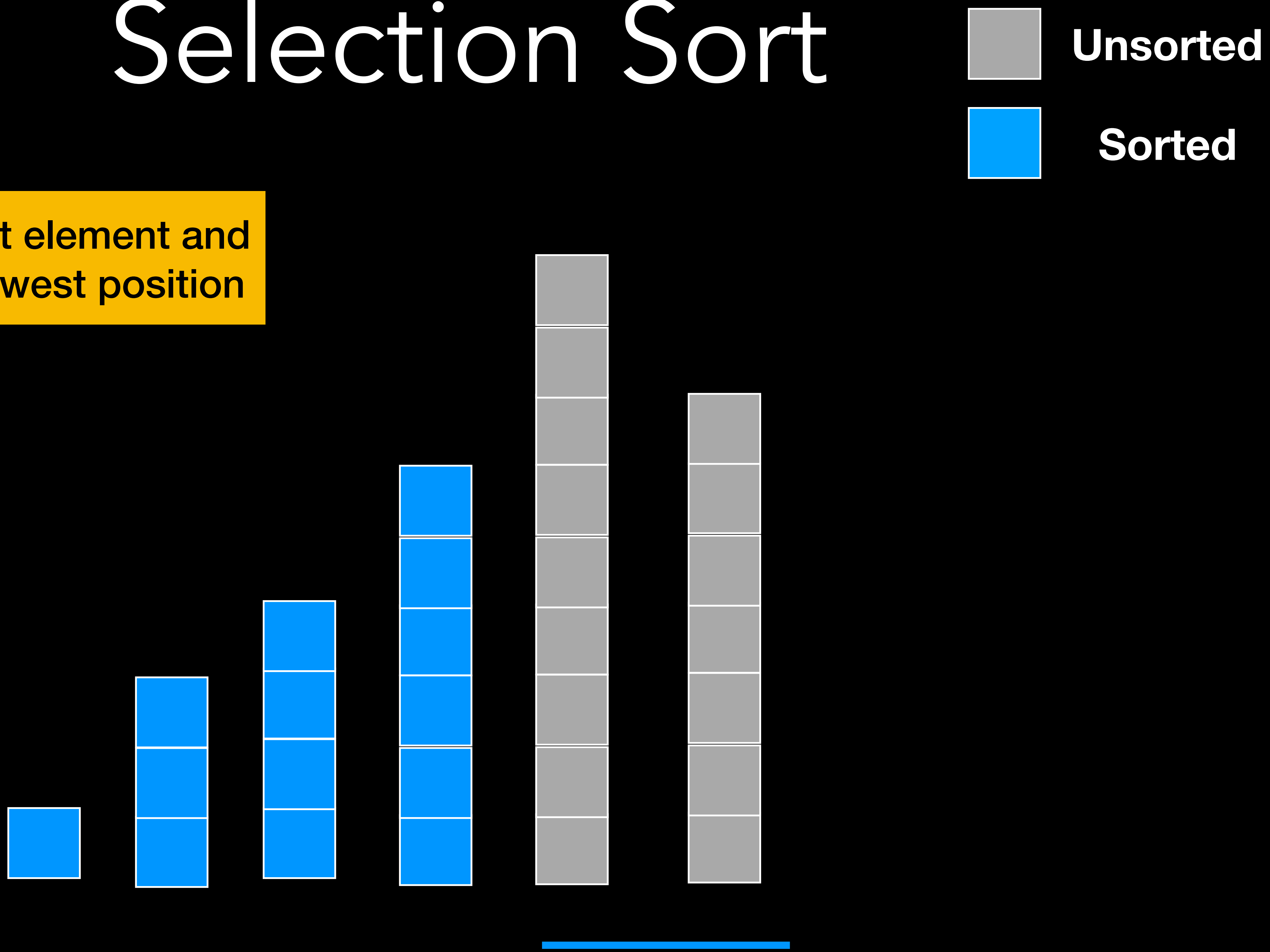
Sorted

Find smallest element and move it at lowest position



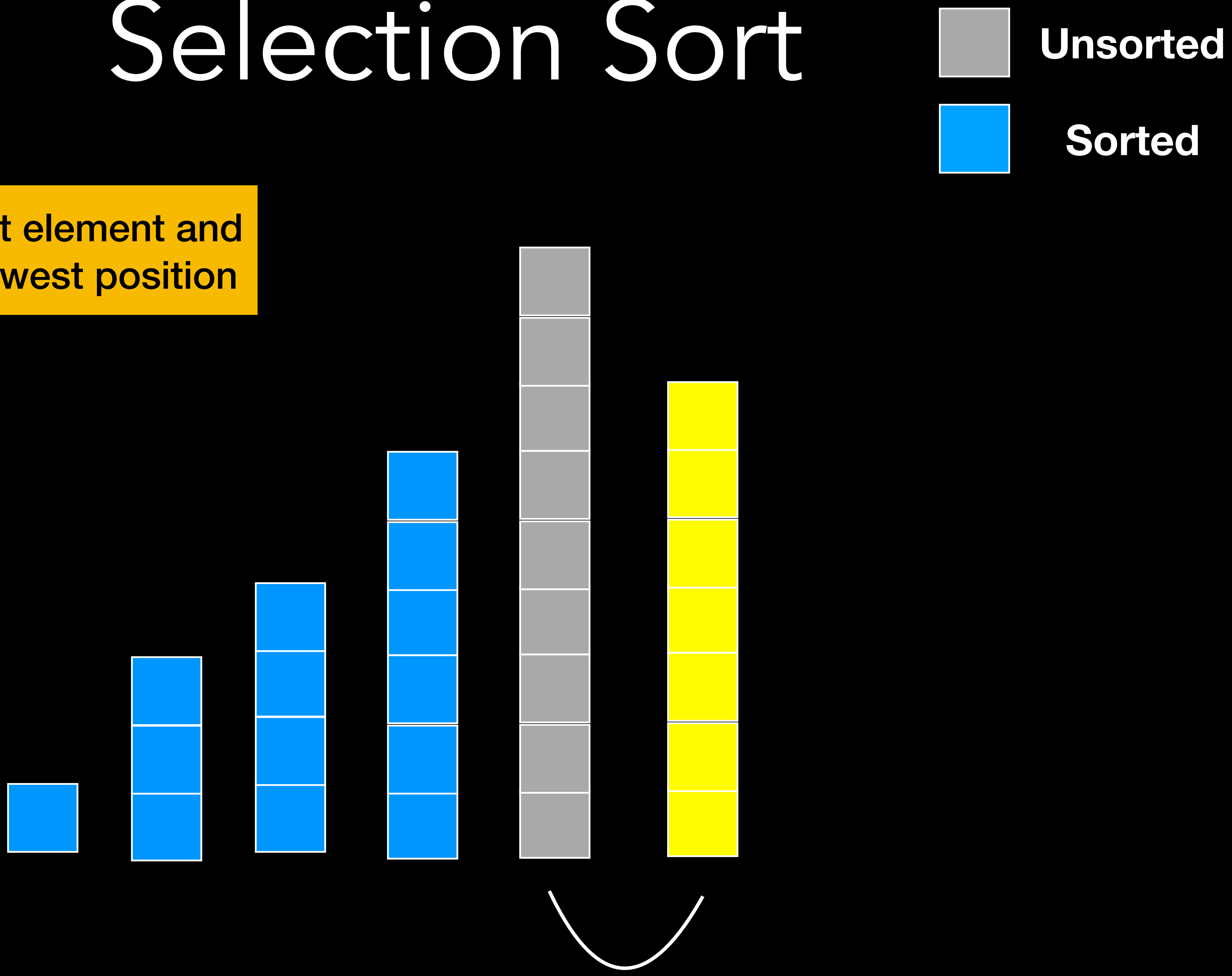
Selection Sort

Find smallest element and move it at lowest position



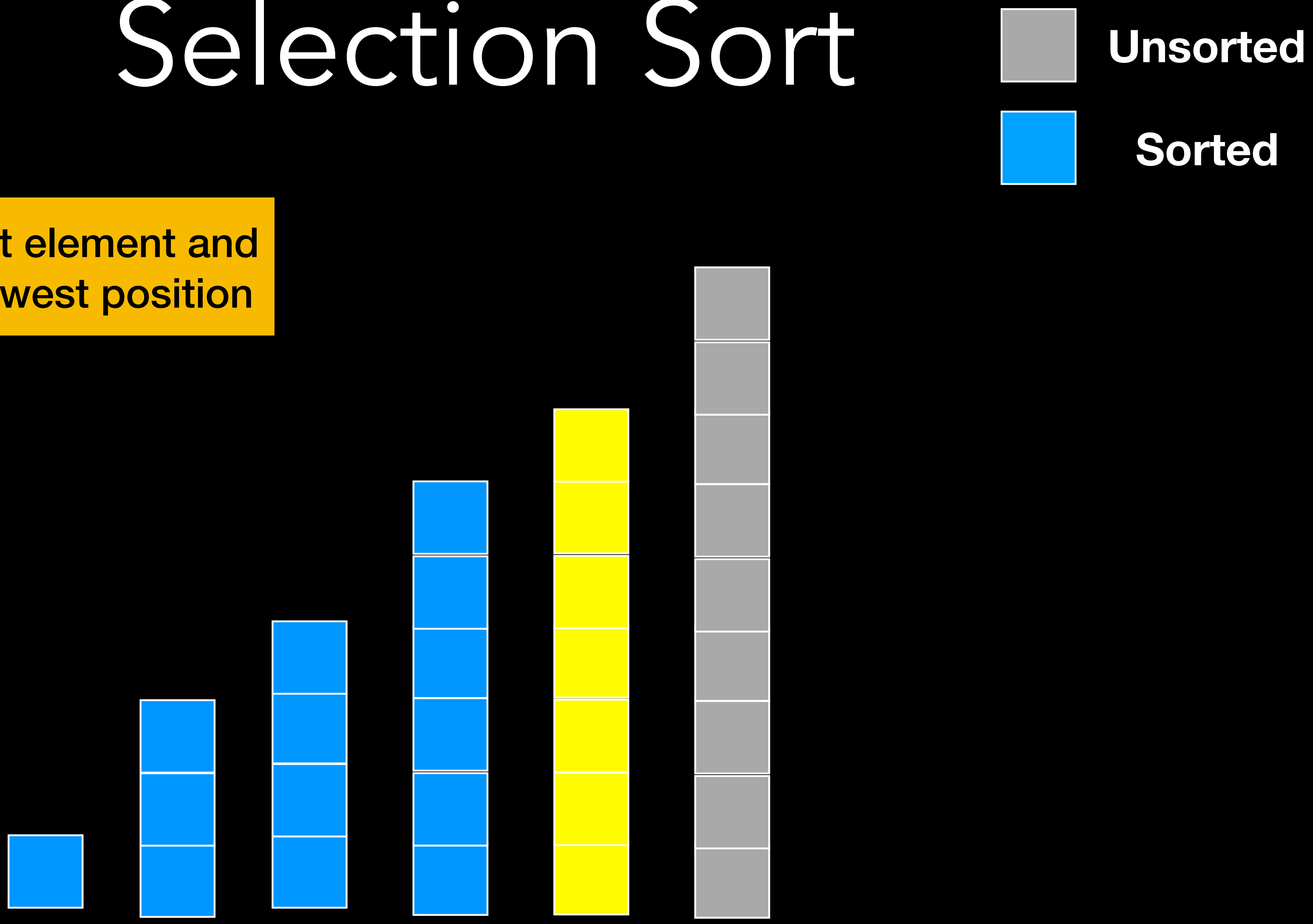
Selection Sort

Find smallest element and move it at lowest position



Selection Sort

Find smallest element and move it at lowest position

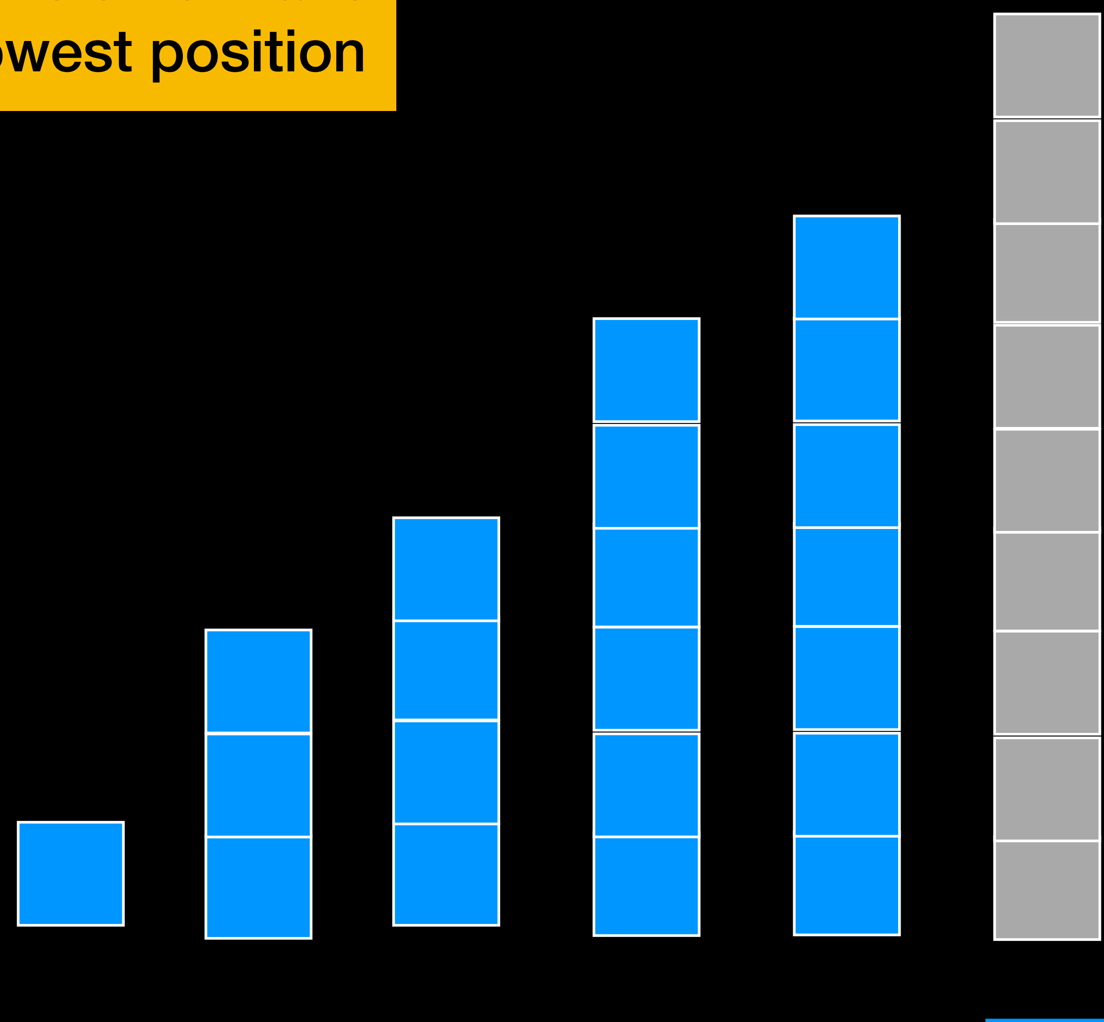


Selection Sort

Unsorted

Sorted

Find smallest element and move it at lowest position

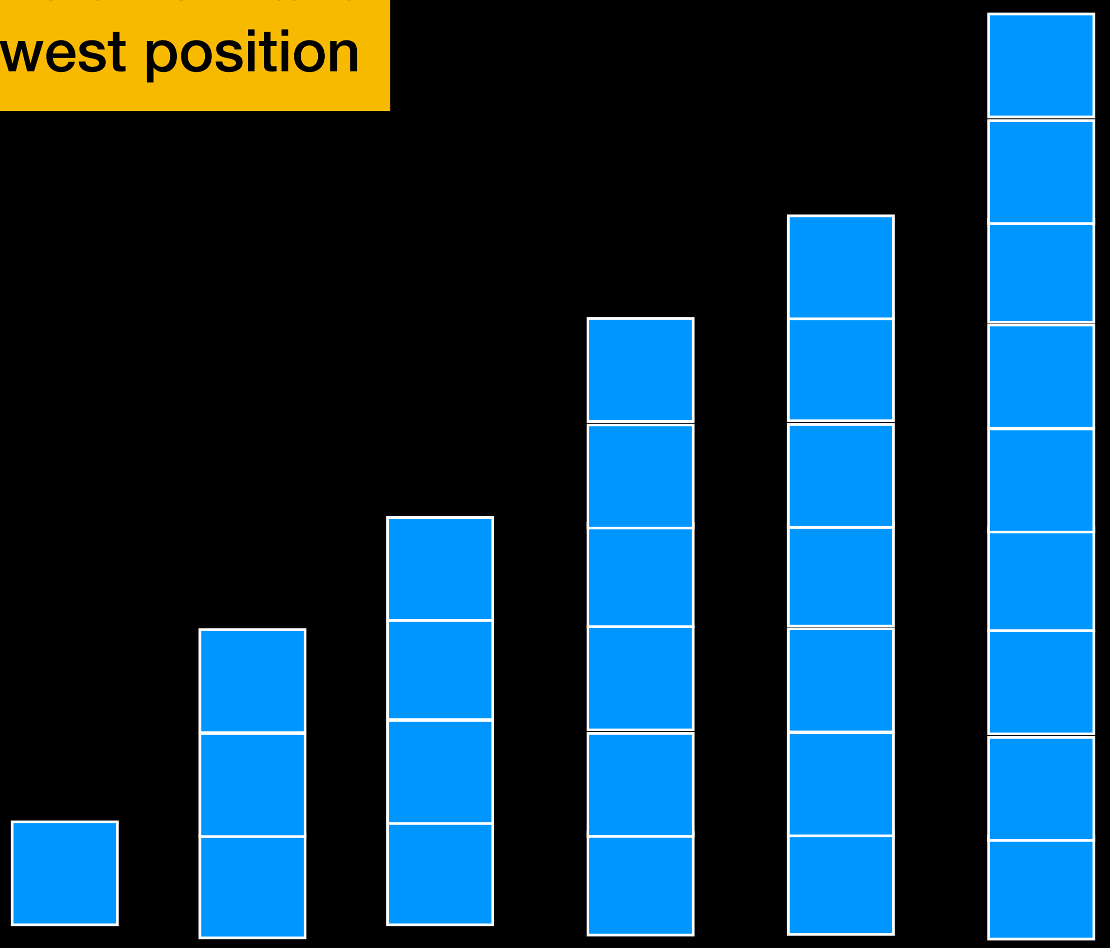


Selection Sort

Unsorted

Sorted

Find smallest element and move it at lowest position



Selection Sort

Find the first 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

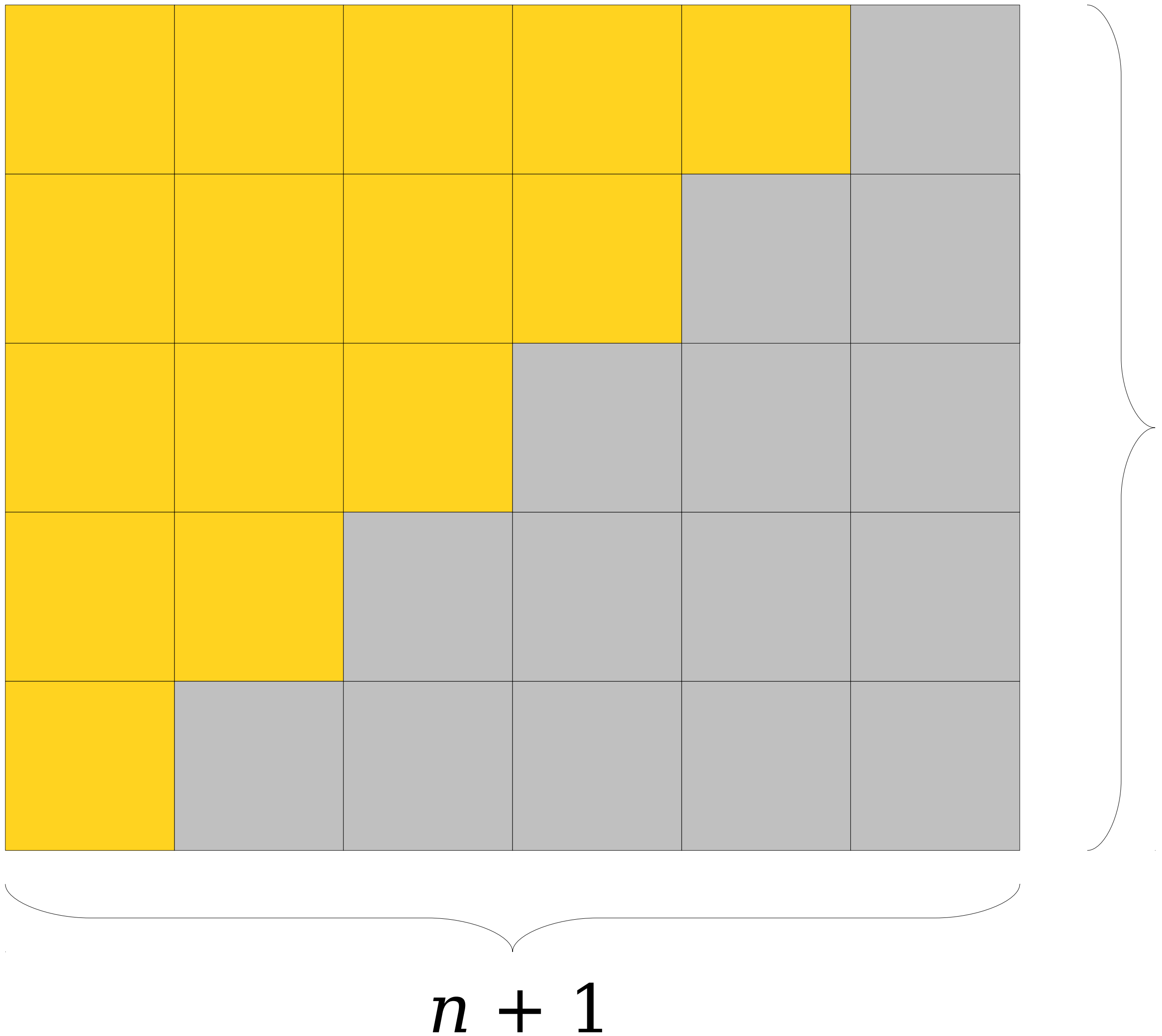
Find second smallest: look at **n-1** elements

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

. . .

Total work: **n + (n-1) + (n-2) + . . . +1**

$n + (n\text{-}1) + \ldots + 2 + 1 = n(n+1) / 2$



Selection Sort Analysis

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

Selection Sort Analysis

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

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

Selection Sort Analysis

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

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

Ignore constant

Ignore non-dominant terms

Selection Sort Analysis

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

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

Ignore constant

Ignore non-dominant terms

Selection Sort Analysis

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

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

Selection Sort run time is $\mathbf{O(n^2)}$

Stability

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

Selection Sort Analysis

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

$O(n^2)$ comparisons

$O(n)$ data moves

Good choice for small **n** and/or data moves are costly

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 ≈ 2 days

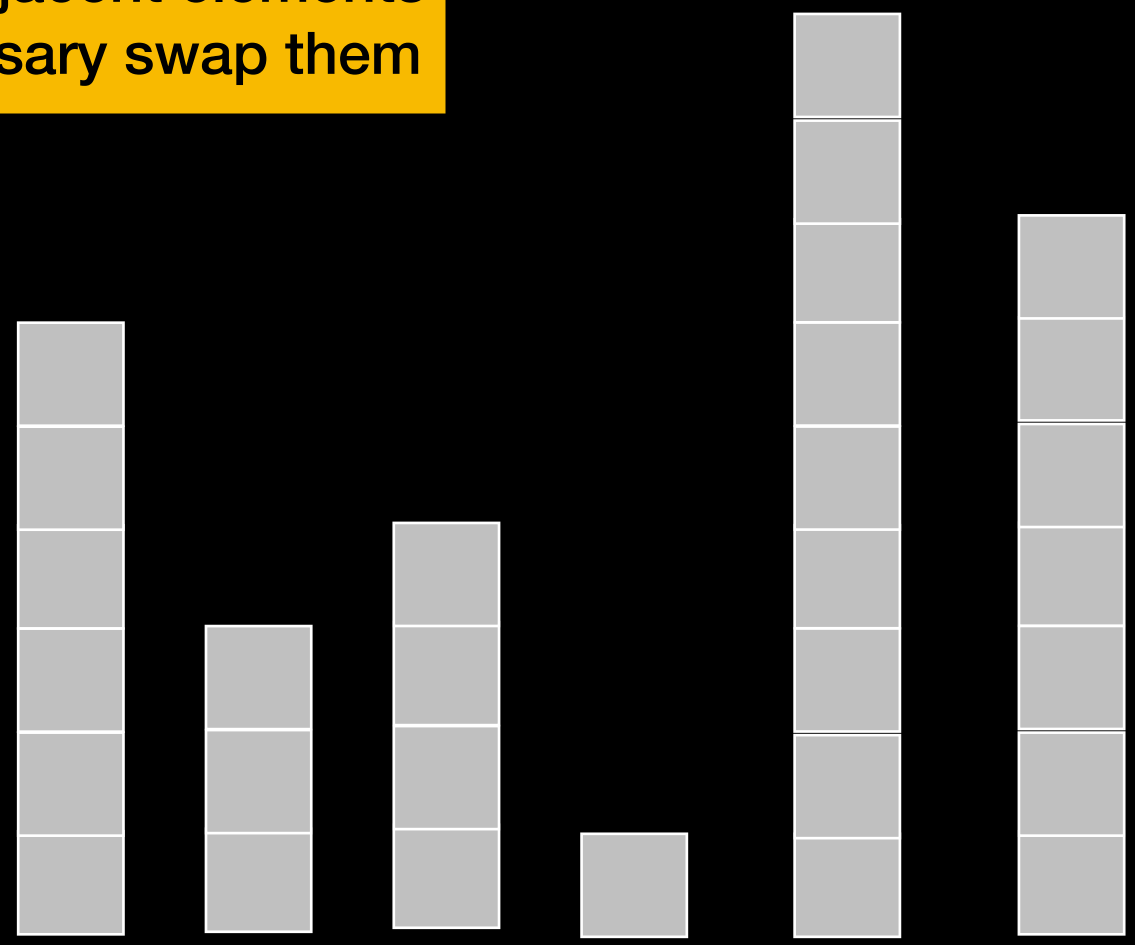
Multiplying input by 100 to go from 17sec to 2 days!!!

Bubble Sort

Unsorted

Sorted

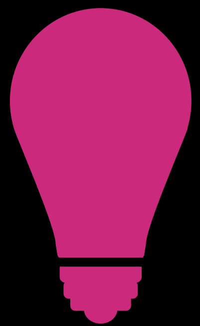
Compare adjacent elements
and if necessary swap them



Bubble Sort

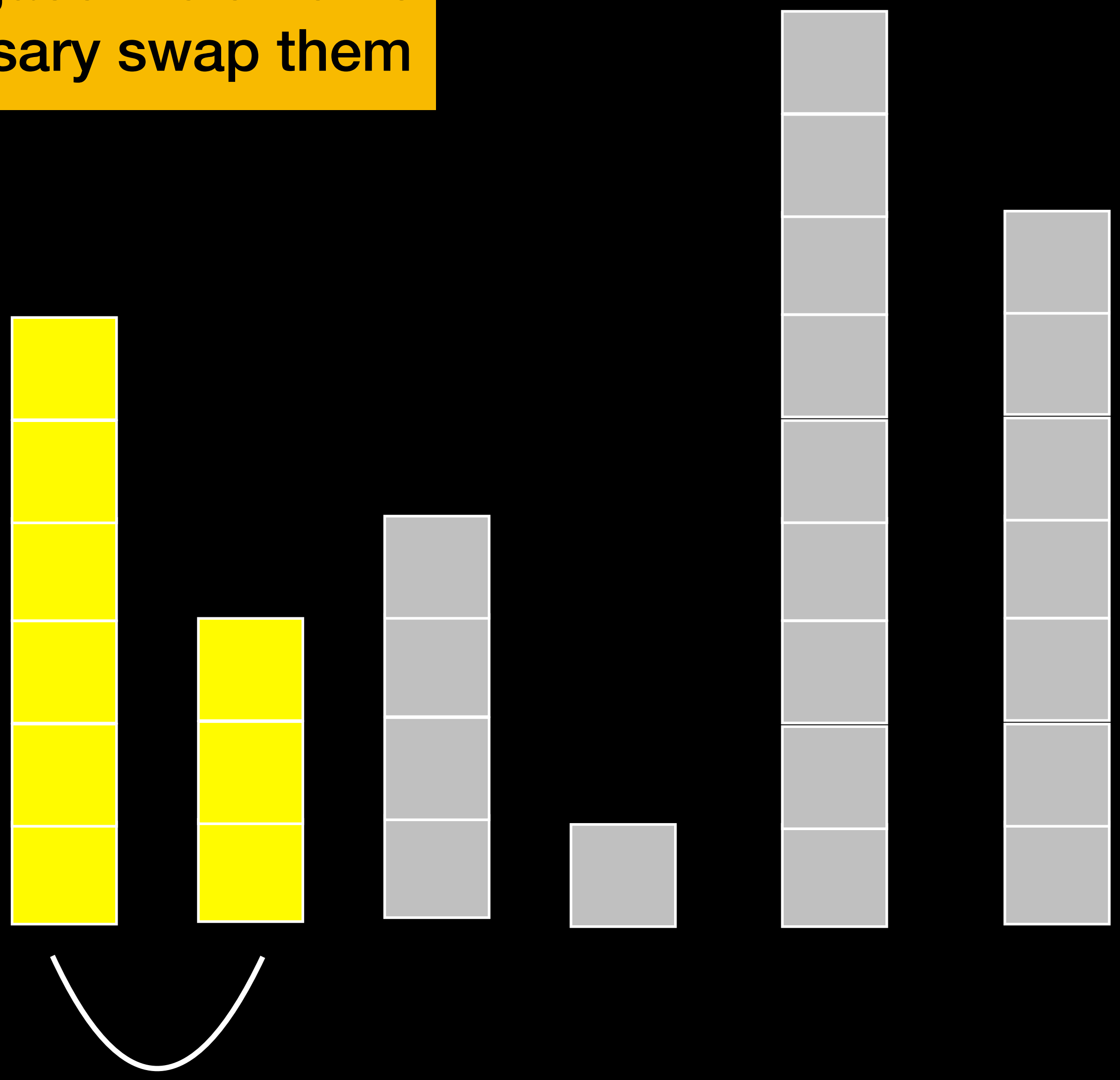
Unsorted

Sorted



Compare adjacent elements and if necessary swap them

1st Pass



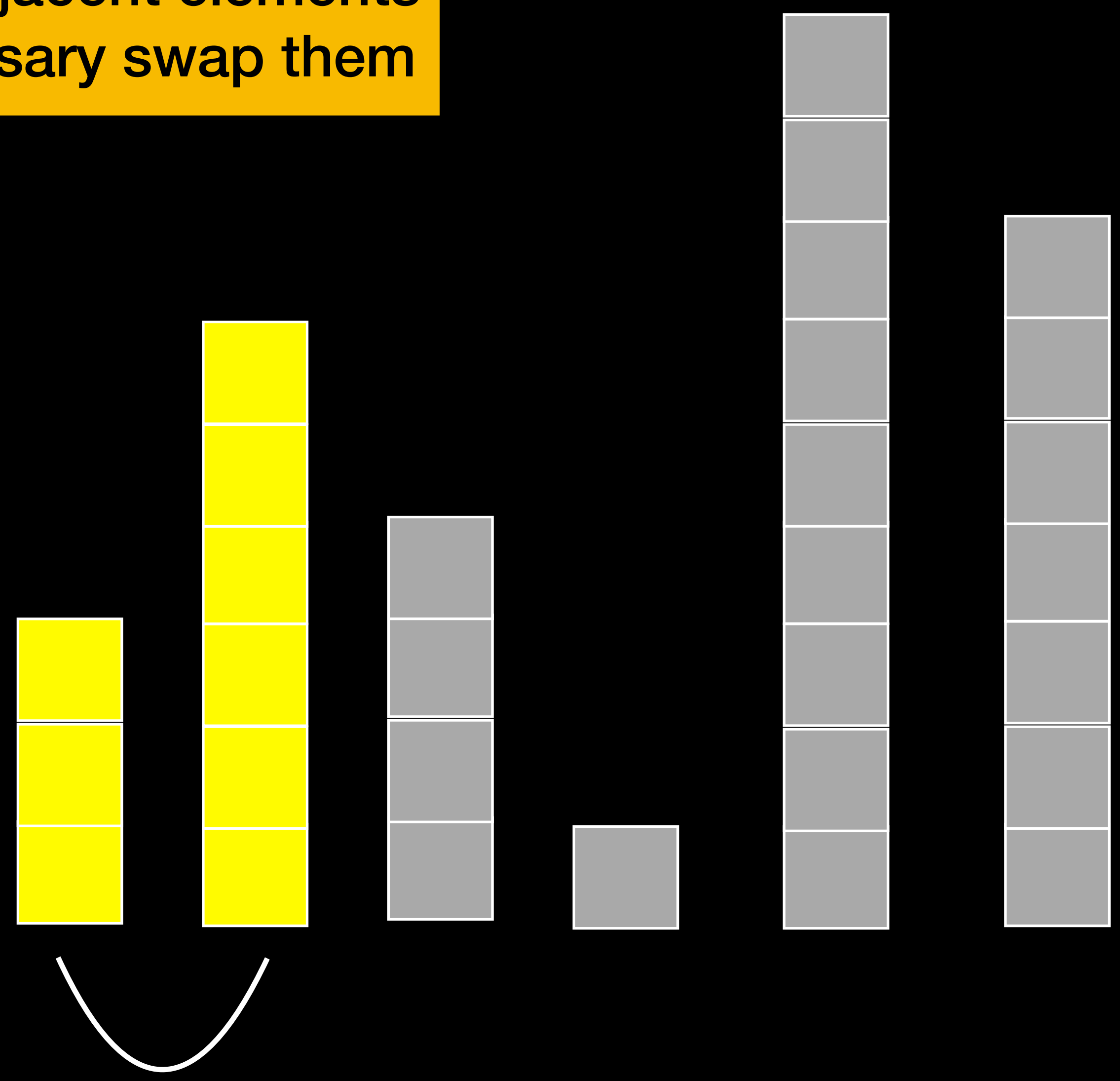
Bubble Sort

Unsorted

Sorted

Compare adjacent elements and if necessary swap them

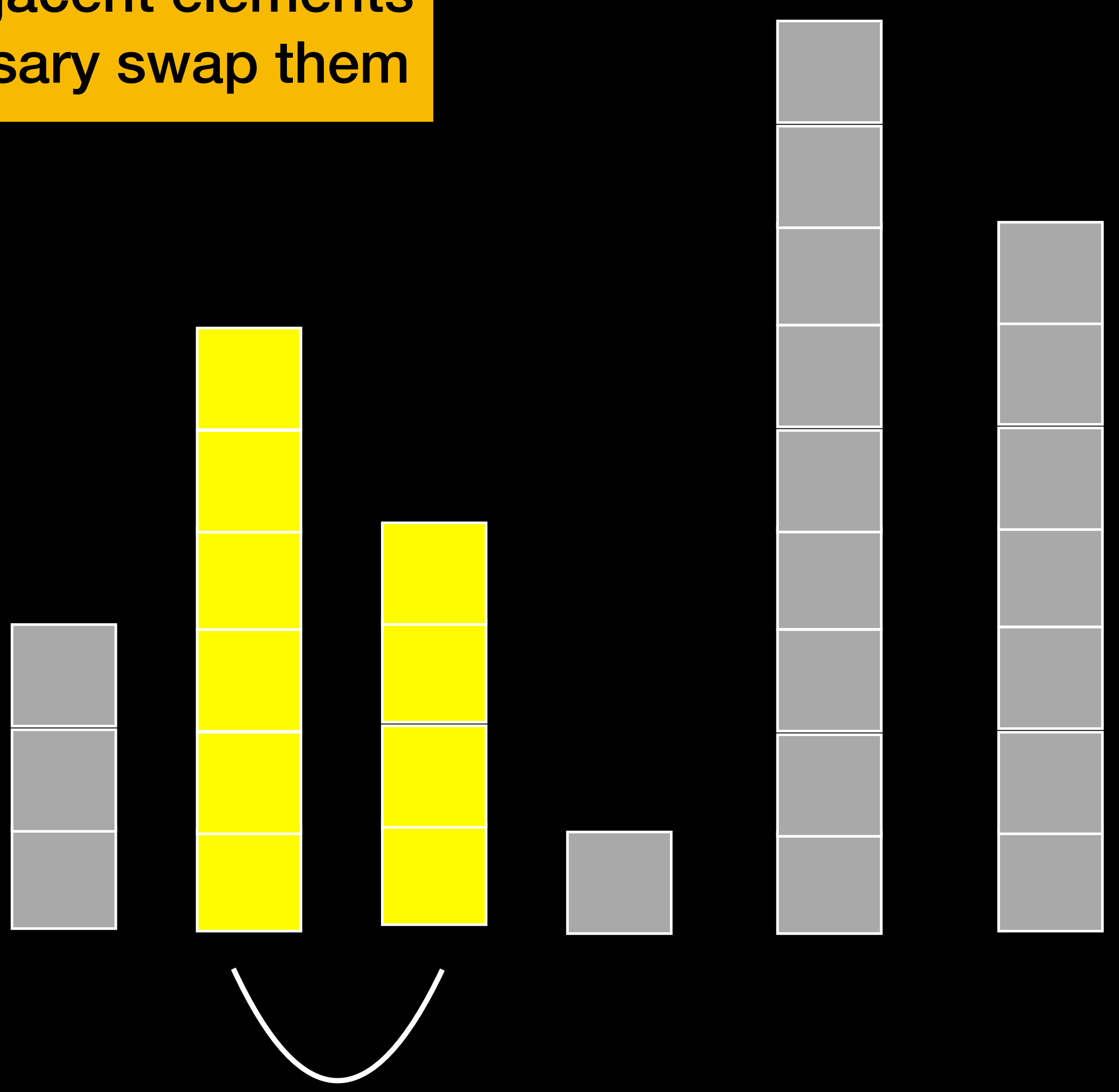
1st Pass



Bubble Sort

Compare adjacent elements and if necessary swap them

1st Pass



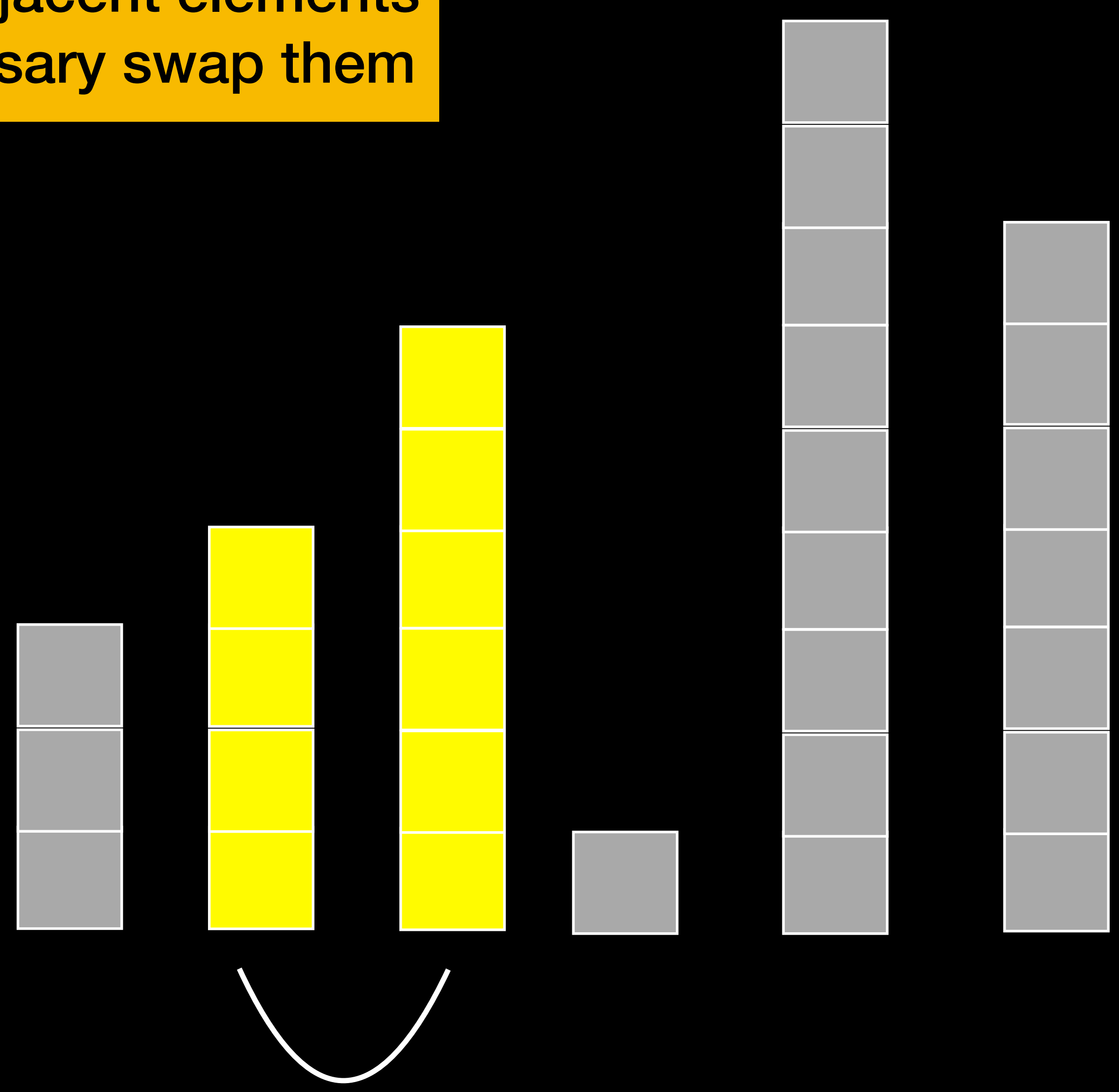
Bubble Sort

Compare adjacent elements and if necessary swap them

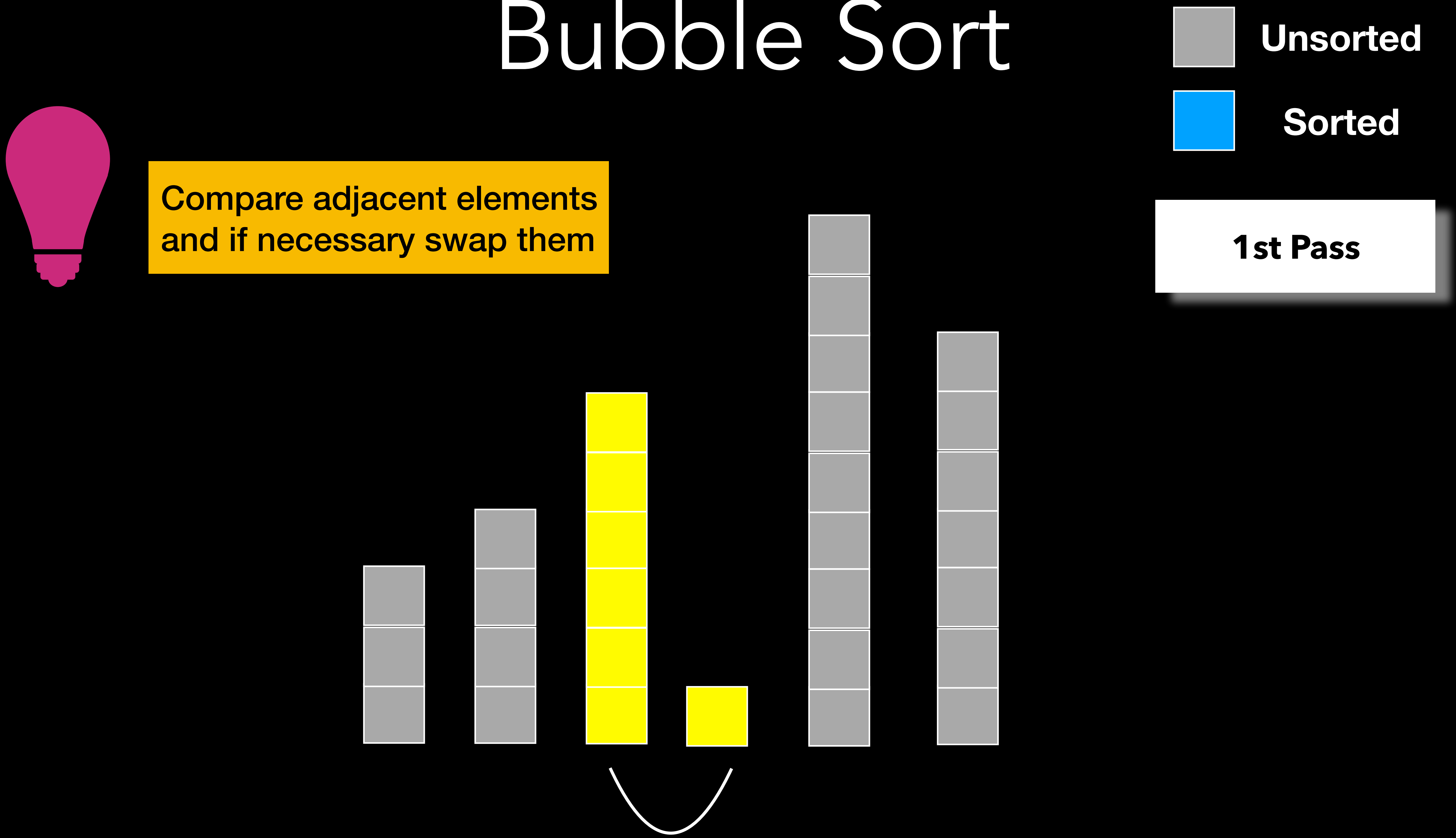
Unsorted

Sorted

1st Pass



Bubble Sort


Unsorted
Sorted
Compare adjacent elements and if necessary swap them
1st Pass

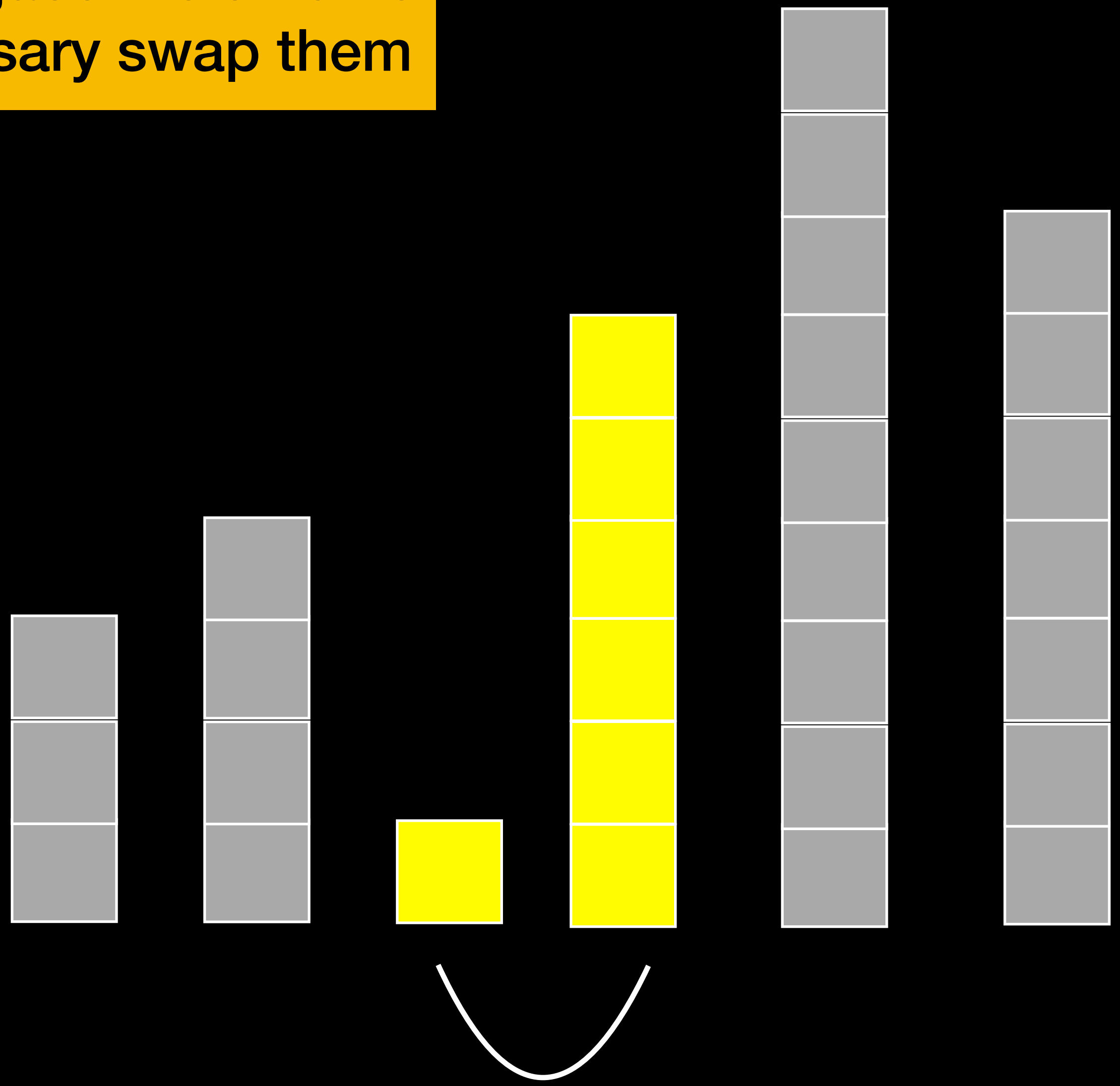
Bubble Sort

Unsorted

Sorted

Compare adjacent elements and if necessary swap them

1st Pass



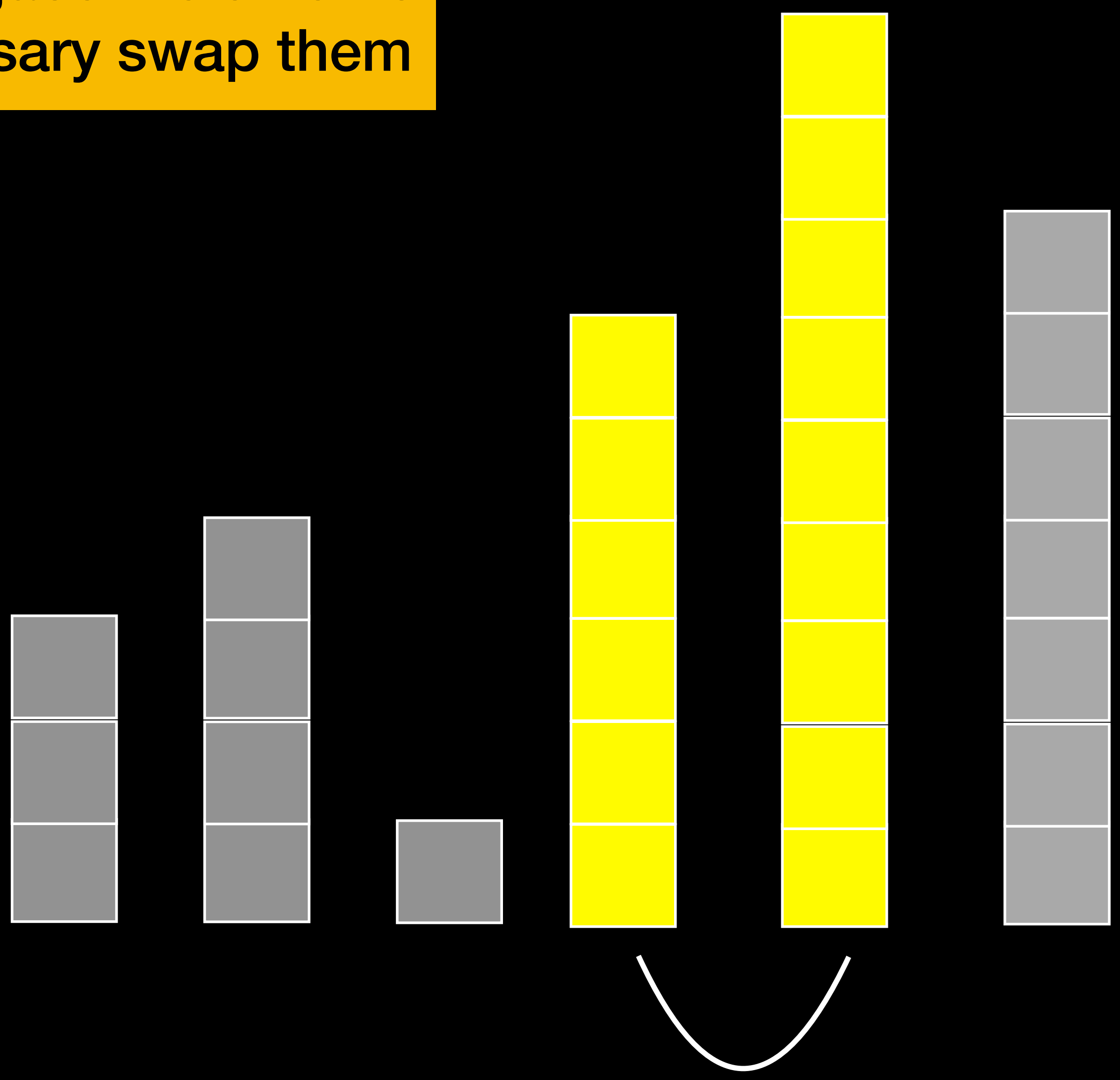
Bubble Sort

Unsorted

Sorted

Compare adjacent elements and if necessary swap them

1st Pass



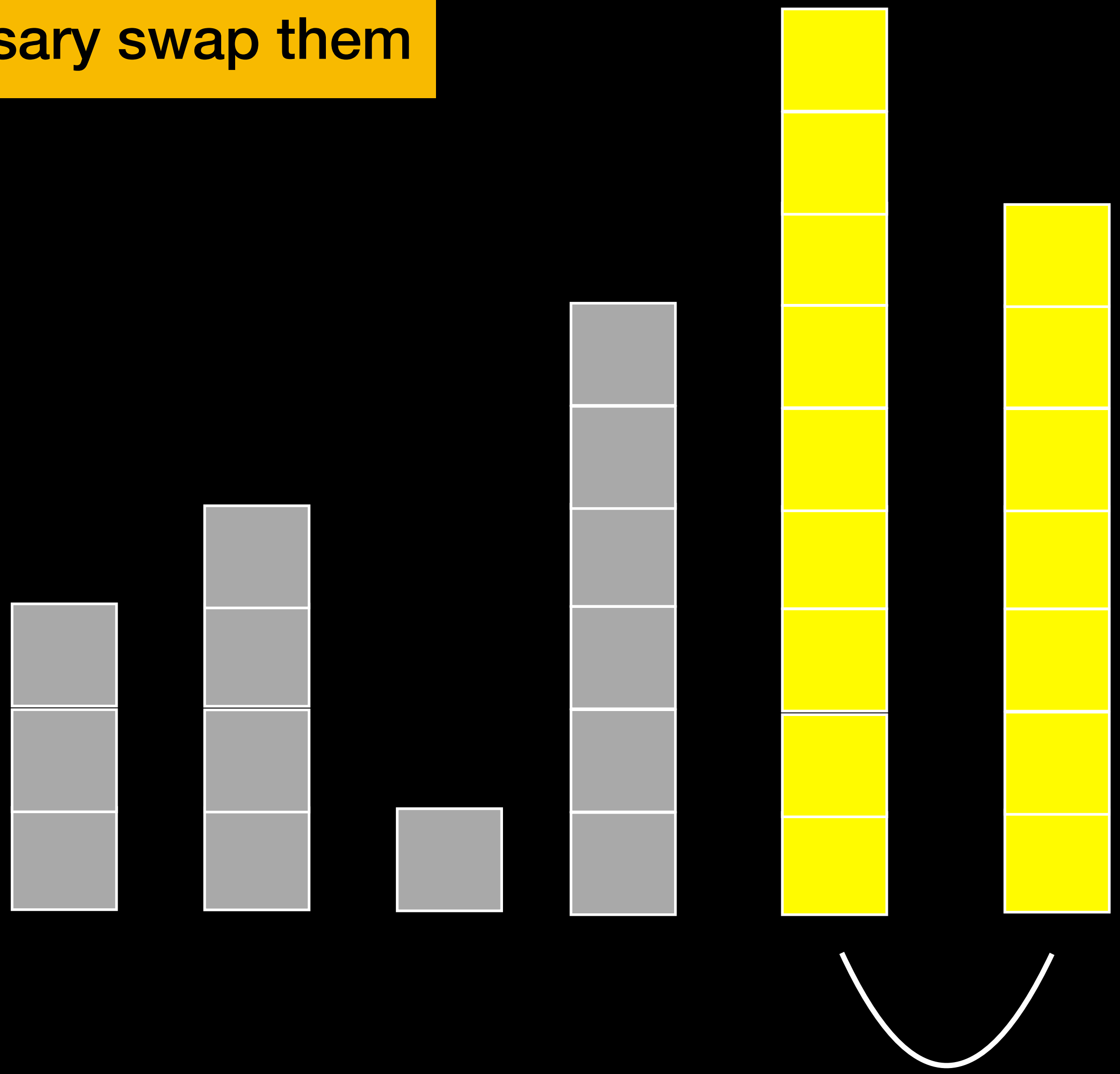
Bubble Sort

Unsorted

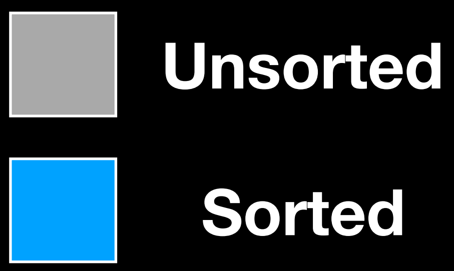
Sorted

Compare adjacent elements and if necessary swap them

1st Pass

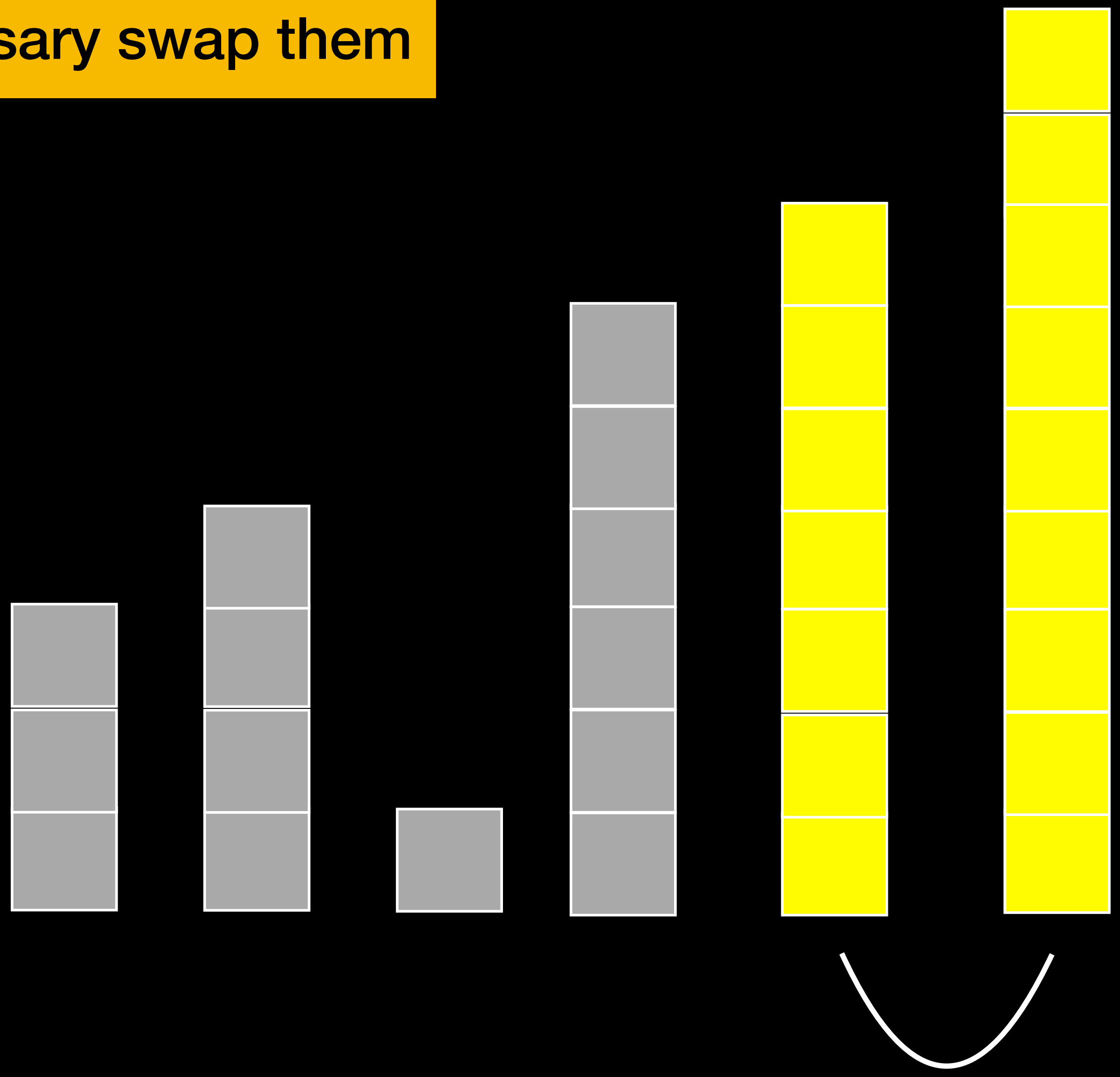


Bubble Sort



Compare adjacent elements and if necessary swap them

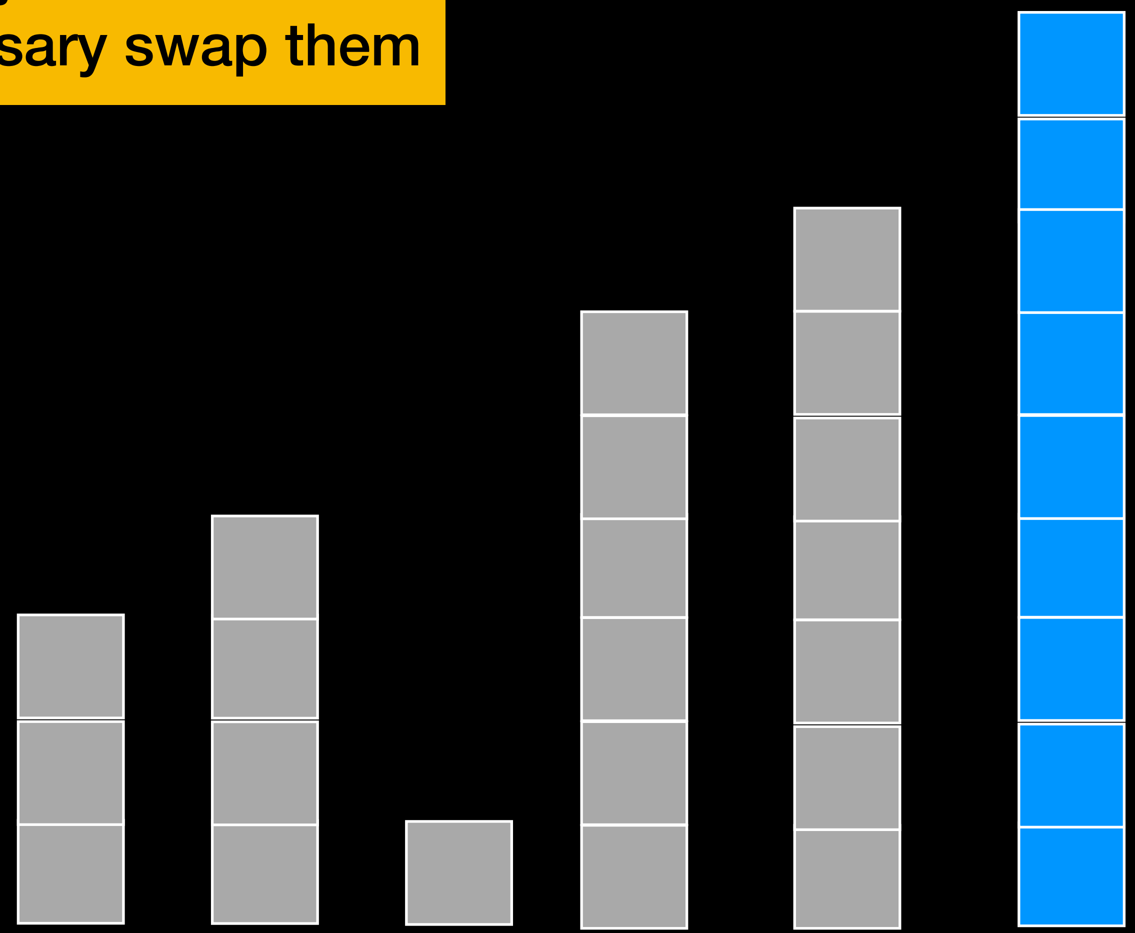
1st Pass



Bubble Sort

Compare adjacent elements and if necessary swap them

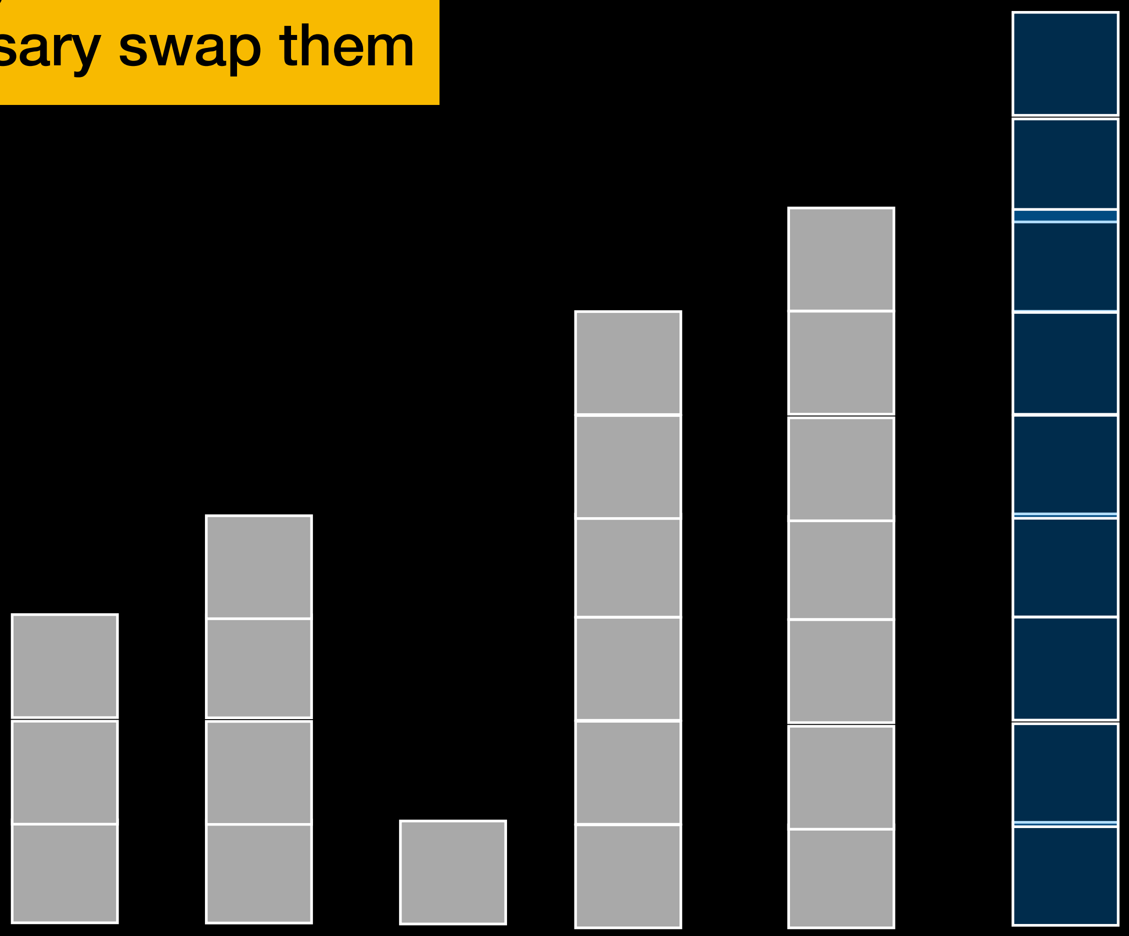
End of1st Pass:
Not sorted, but largest has ***"bubbled up"*** to its proper position



Bubble Sort

Compare adjacent elements and if necessary swap them

2nd Pass:
Sort **n-1**
… and so on



Bubble Sort Analysis

How much work?

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

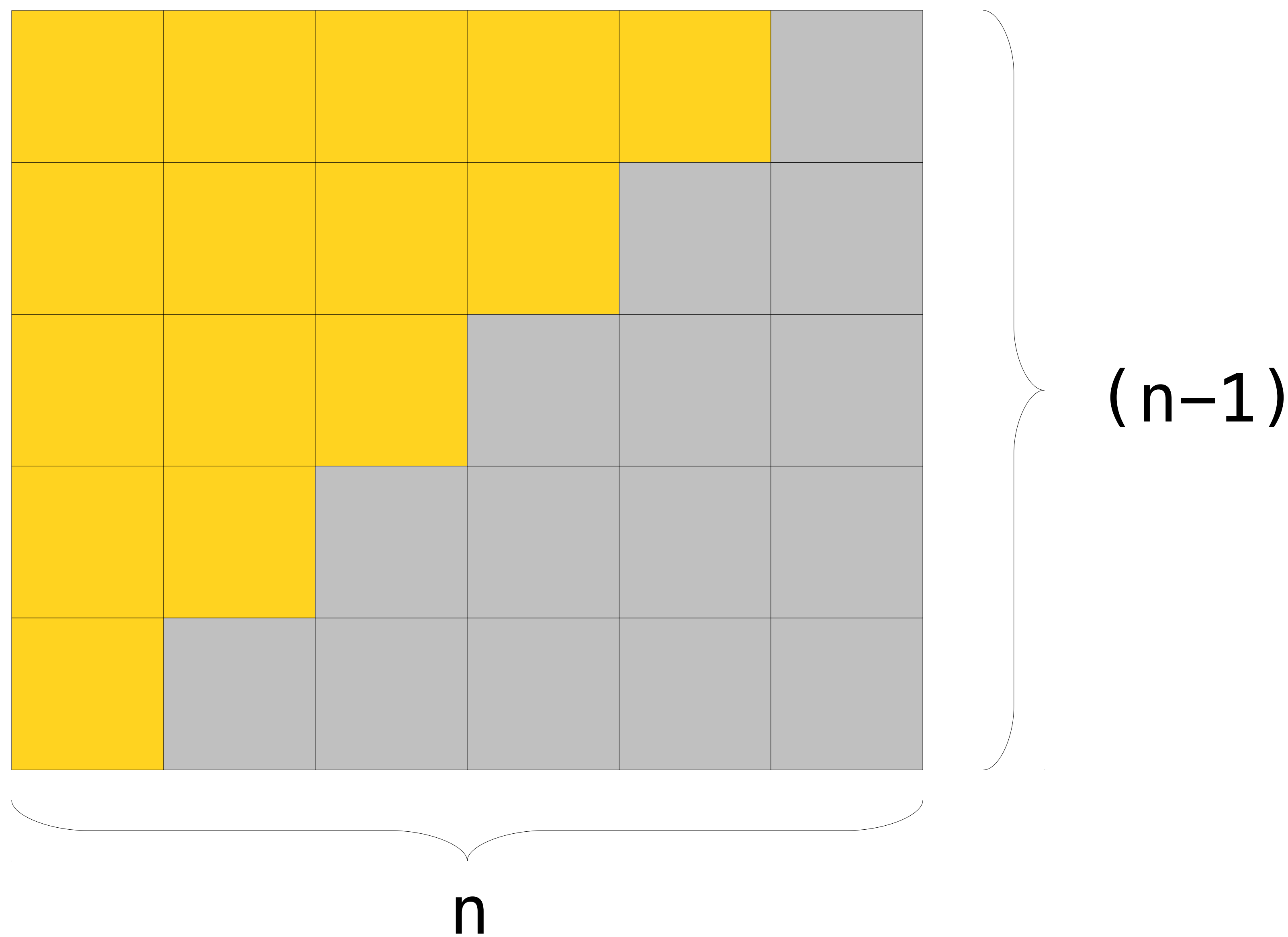
Second pass: **n-2** comparisons and at most **n-2** swaps

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

. . .

Total work: **(n-1) + (n-2) + . . . +1**

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



Bubble Sort Analysis

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

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

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

Bubble Sort Analysis

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

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

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

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

Bubble Sort Analysis

$T(n)$ = n(n-1) / 2 comparisons + n(n-1) / 2 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

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

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

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

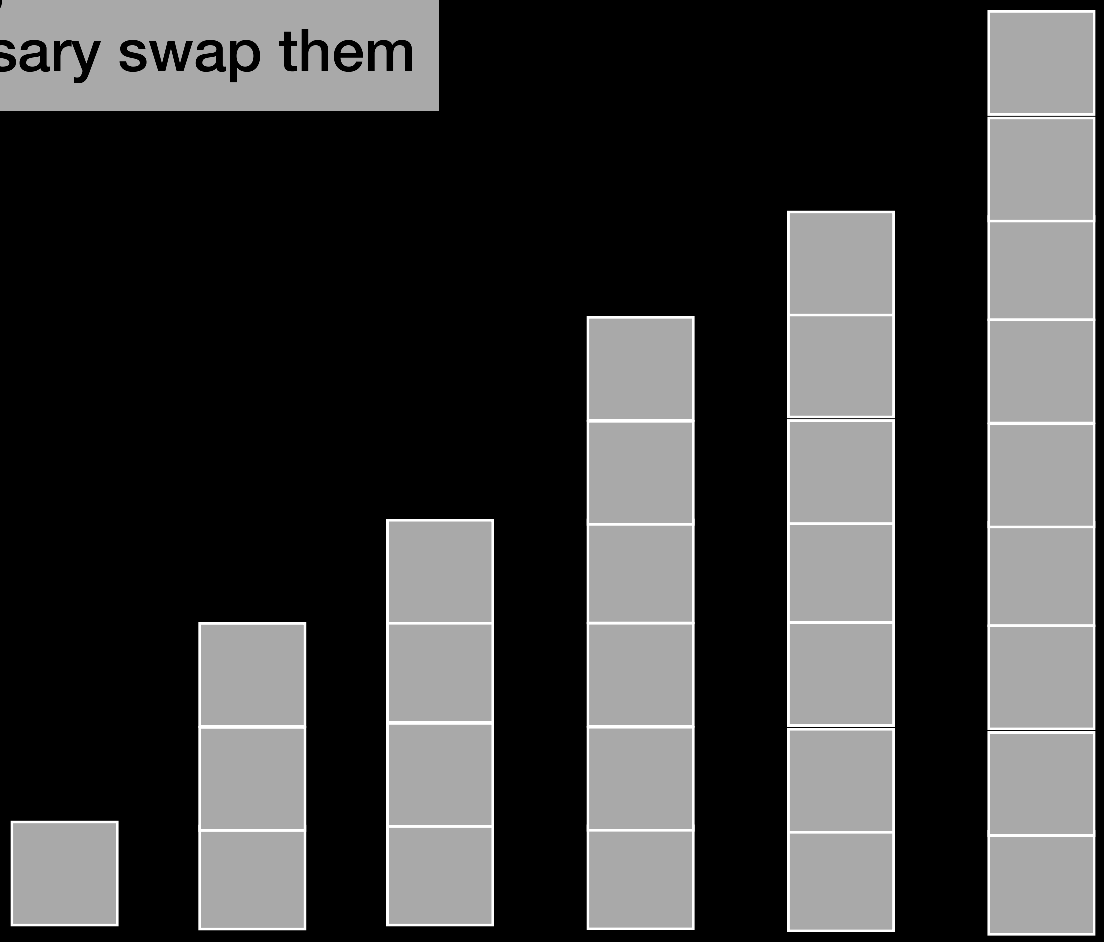
$T(n) = 2(\ (n^2-n) / 2\ ) = O(\ )$?

$T(n) = n^2-n = O(\ \mathbf{n^2})$

Bubble Sort run time is $\mathbf{O(\ n^2)}$

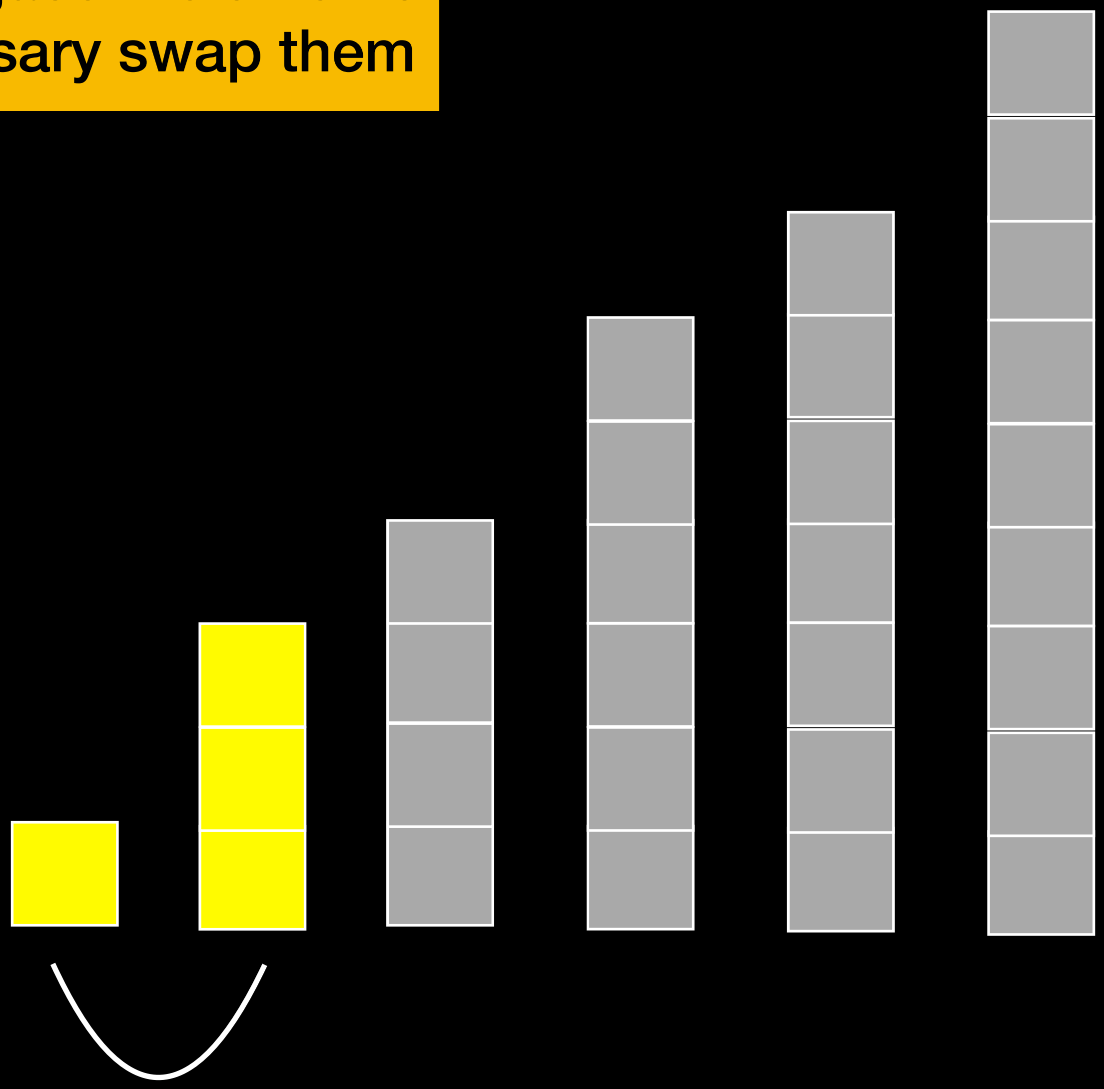
Bubble Sort

Compare adjacent elements and if necessary swap them



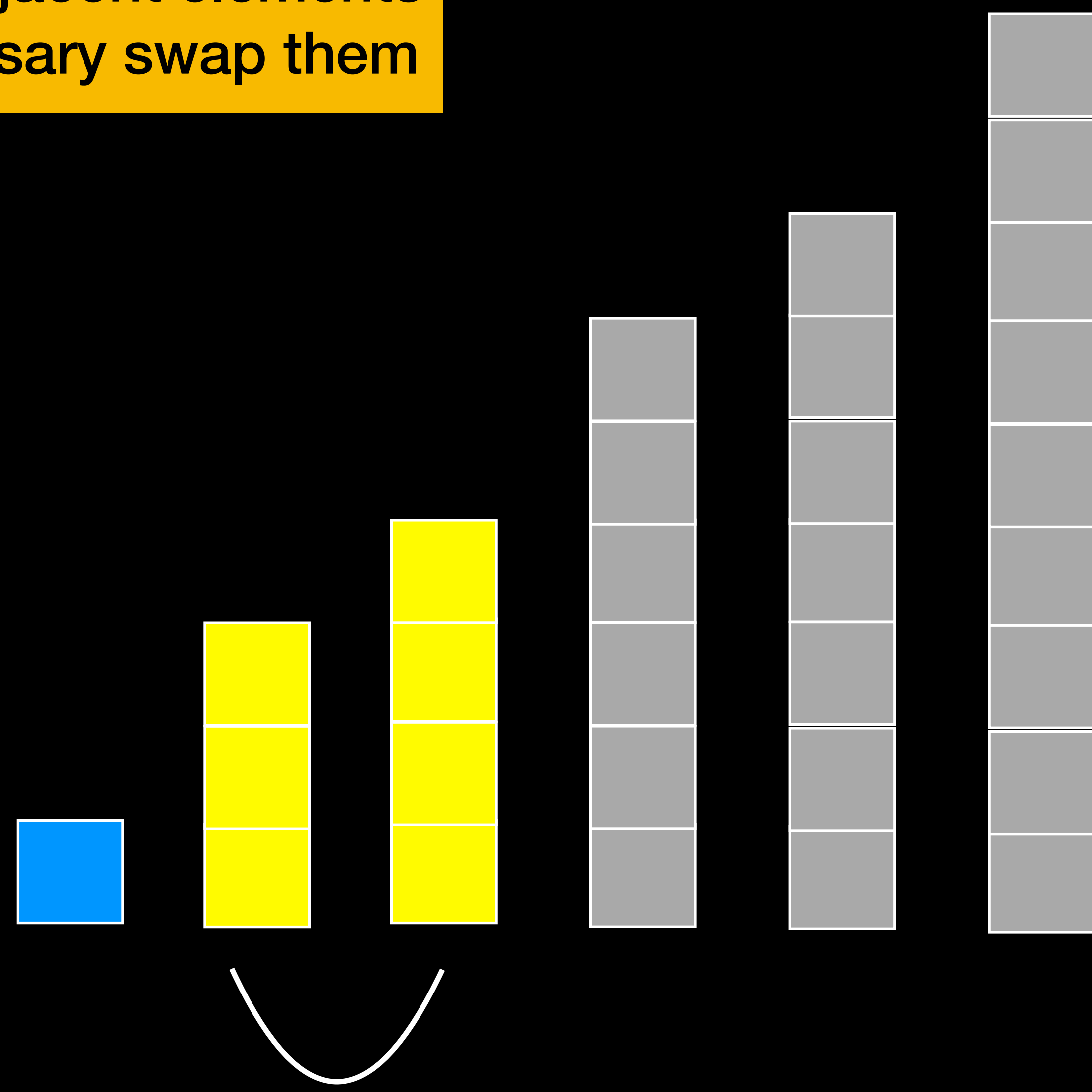
Bubble Sort

Compare adjacent elements
and if necessary swap them



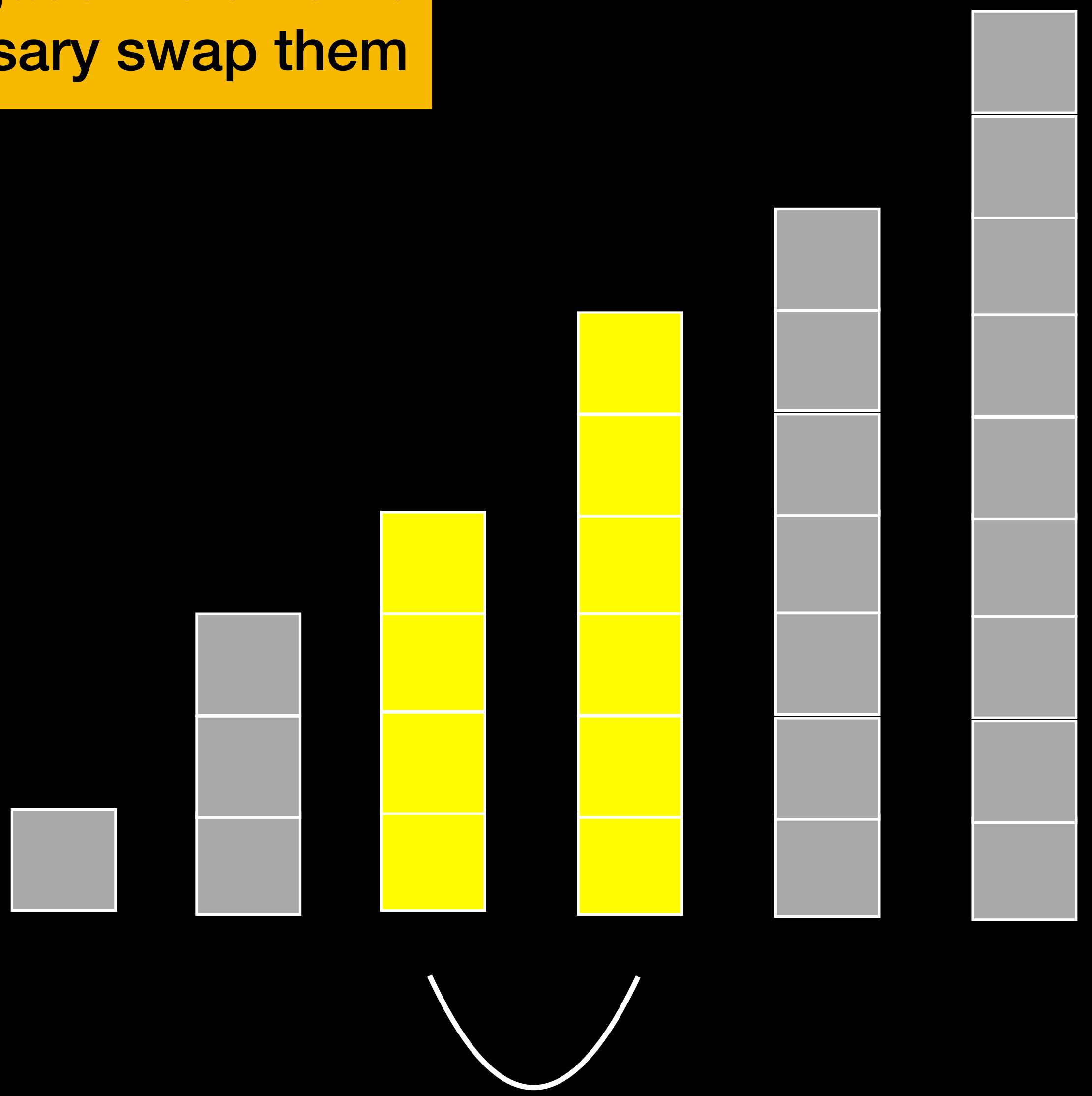
Bubble Sort

Compare adjacent elements and if necessary swap them



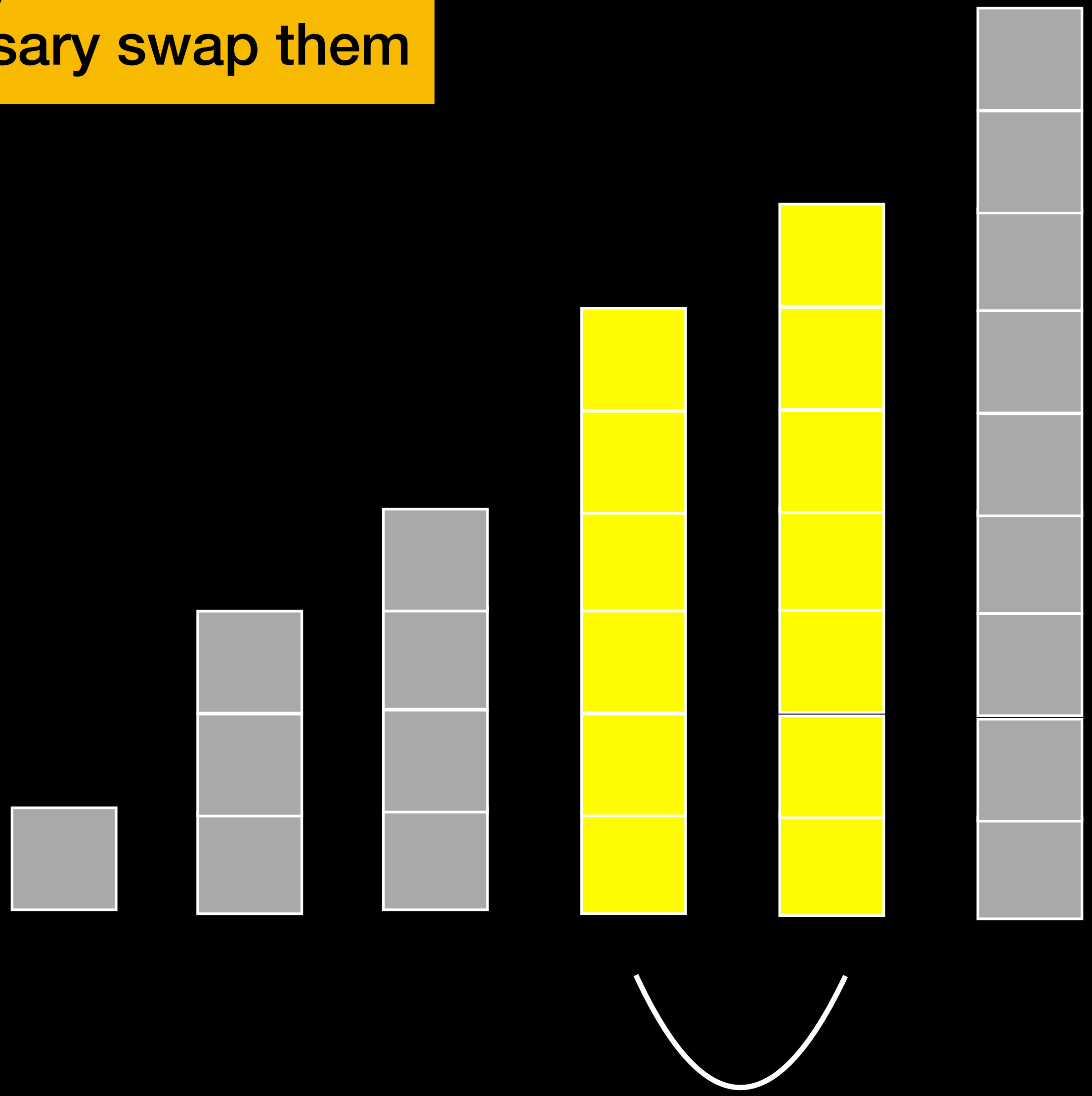
Bubble Sort

Compare adjacent elements and if necessary swap them



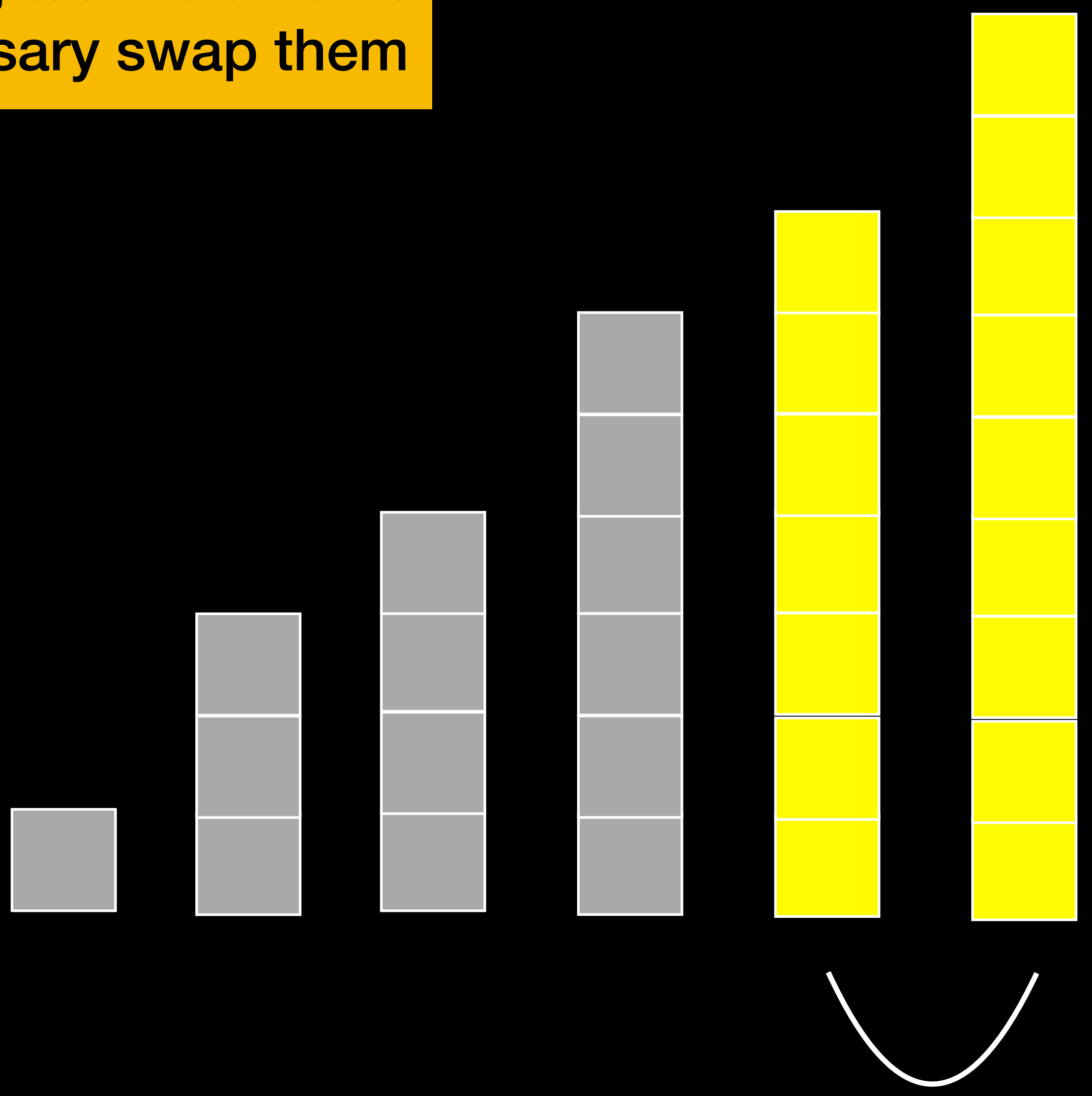
Bubble Sort

Compare adjacent elements and if necessary swap them

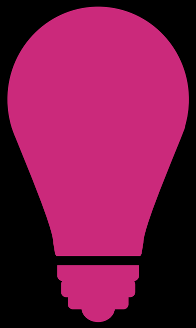


Bubble Sort

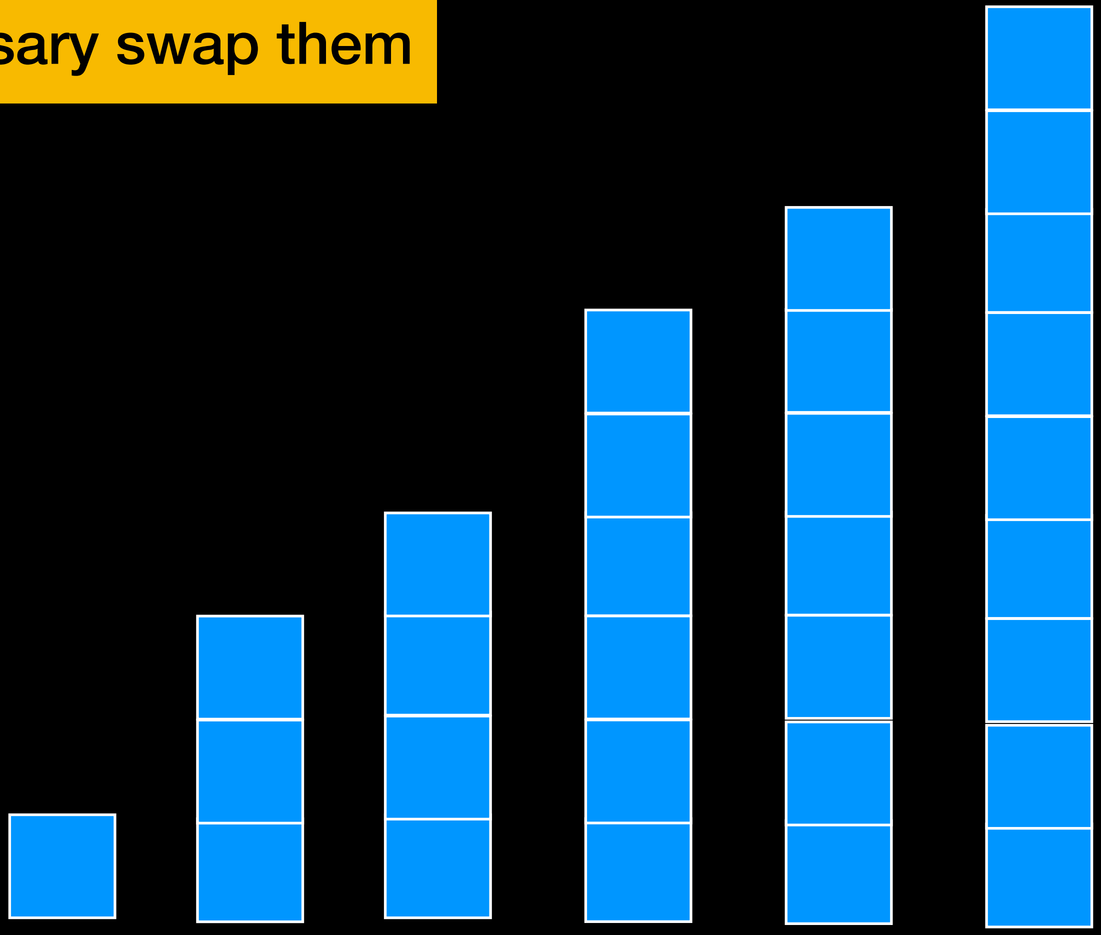
Compare adjacent elements and if necessary swap them



Bubble Sort



Compare adjacent elements and if necessary swap them



Bubble Sort Analysis

Execution time DOES depend on initial arrangement of data

$O(n^2)$ comparisons and data moves

$\Omega(n)$

Stable

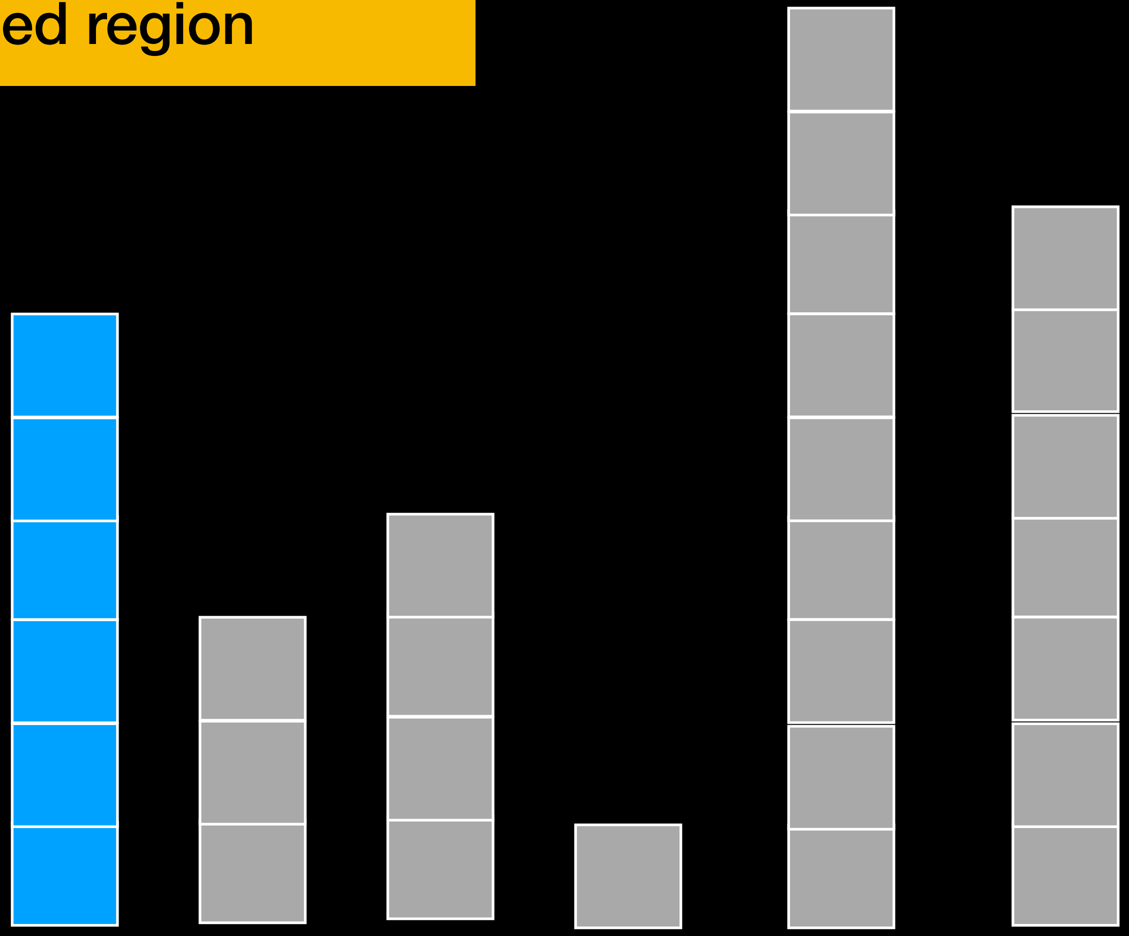
If array is already sorted bubble sort will stop after first pass and no swaps => good choice for small n and data likely somewhat sorted https://www.youtube.com/watch?v=lyZQPjUT5B4

Insertion Sort

Unsorted

Sorted

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

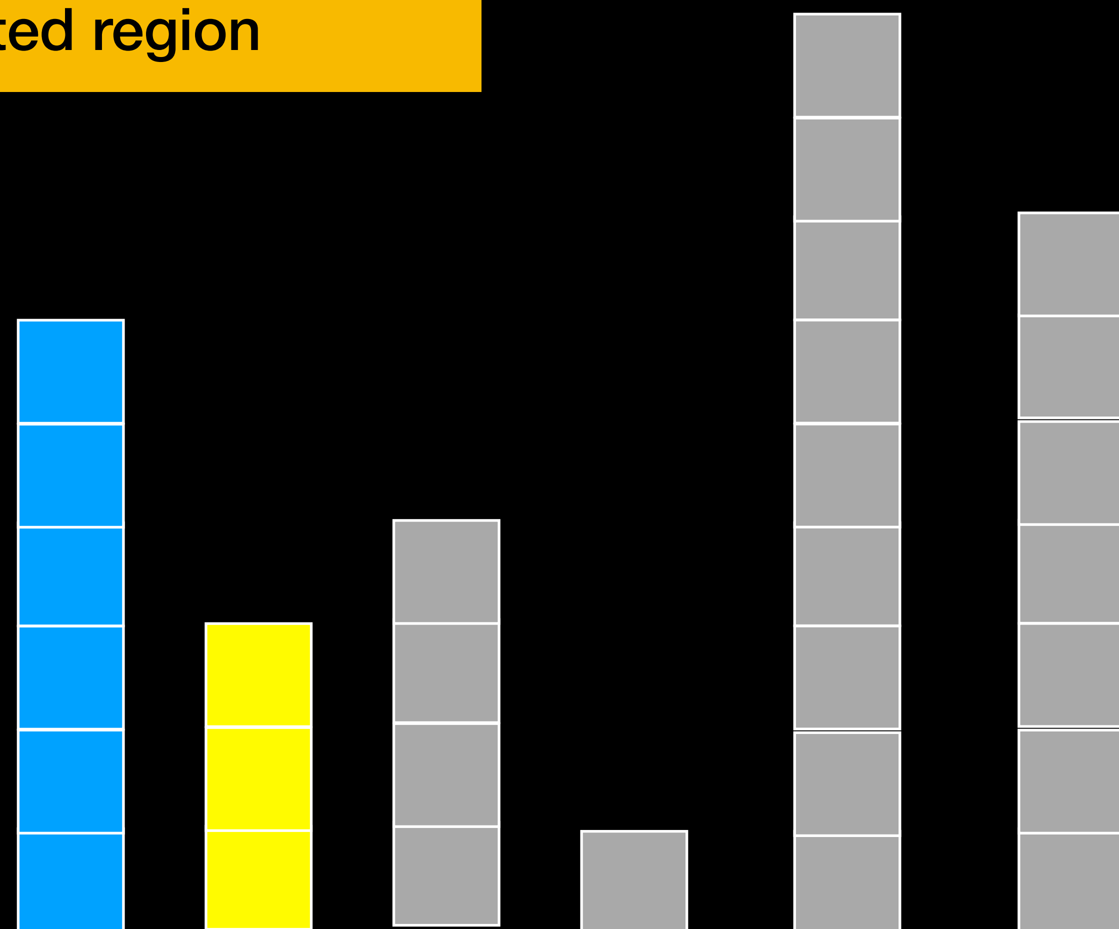


Insertion Sort

Unsorted

Sorted

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

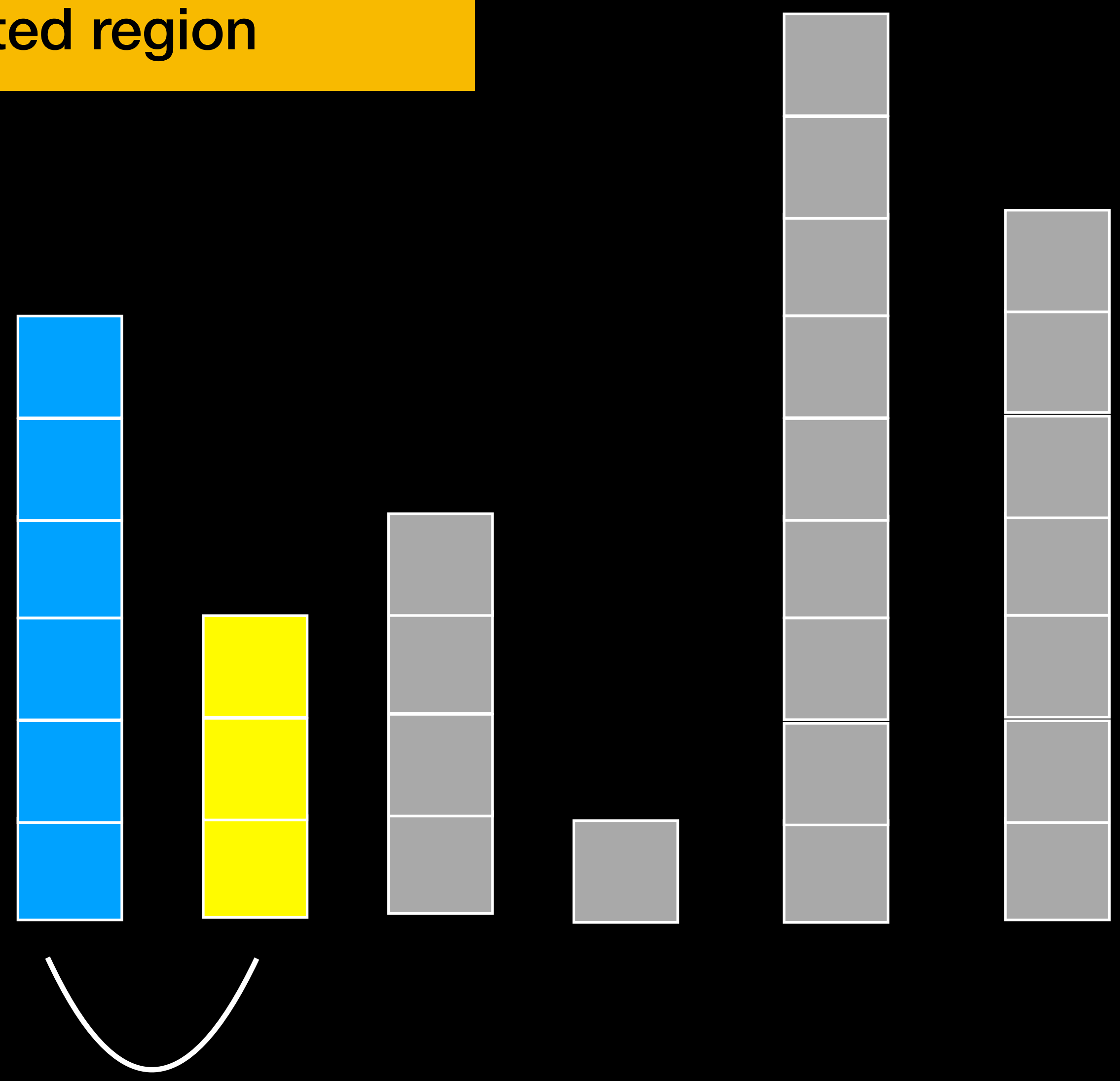


Insertion Sort

Unsorted

Sorted

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

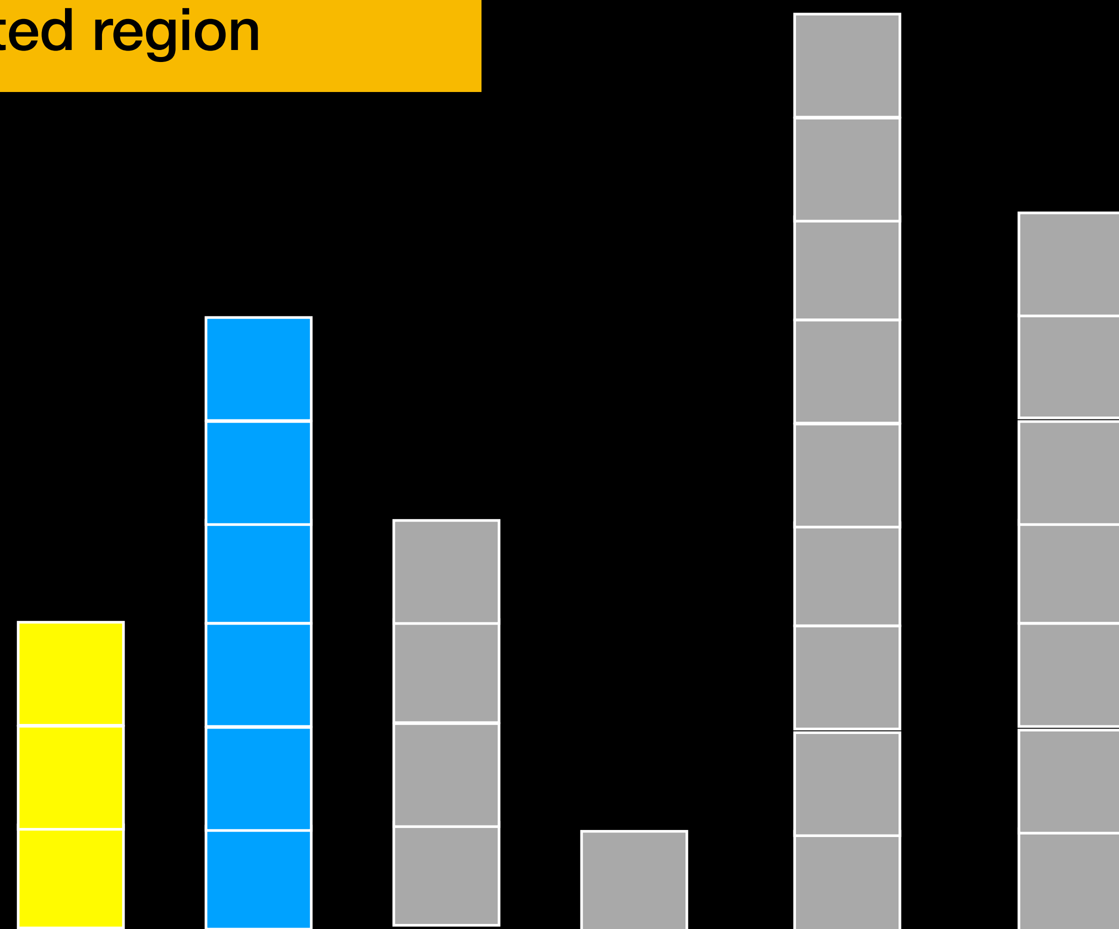


Insertion Sort

Unsorted

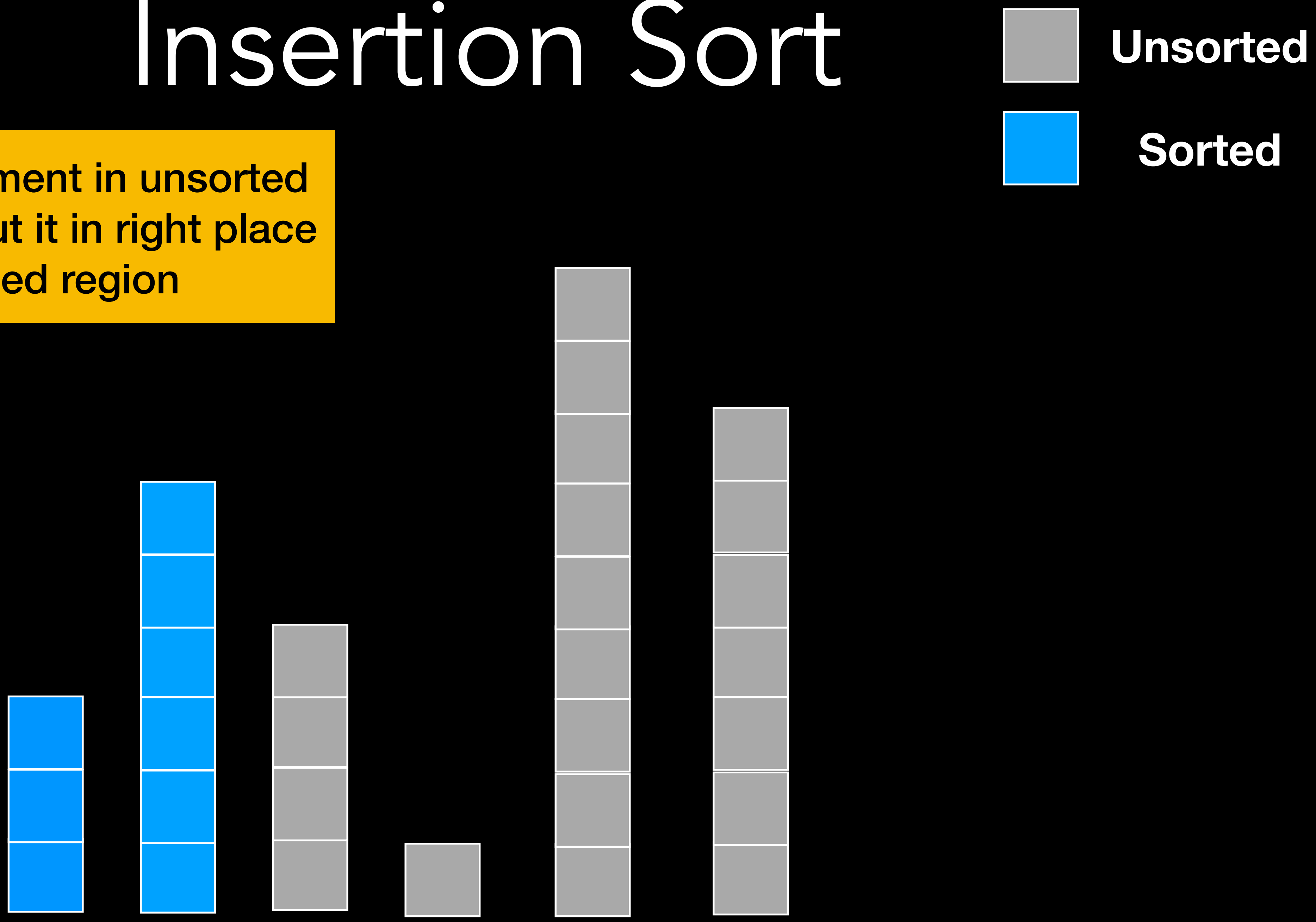
Sorted

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

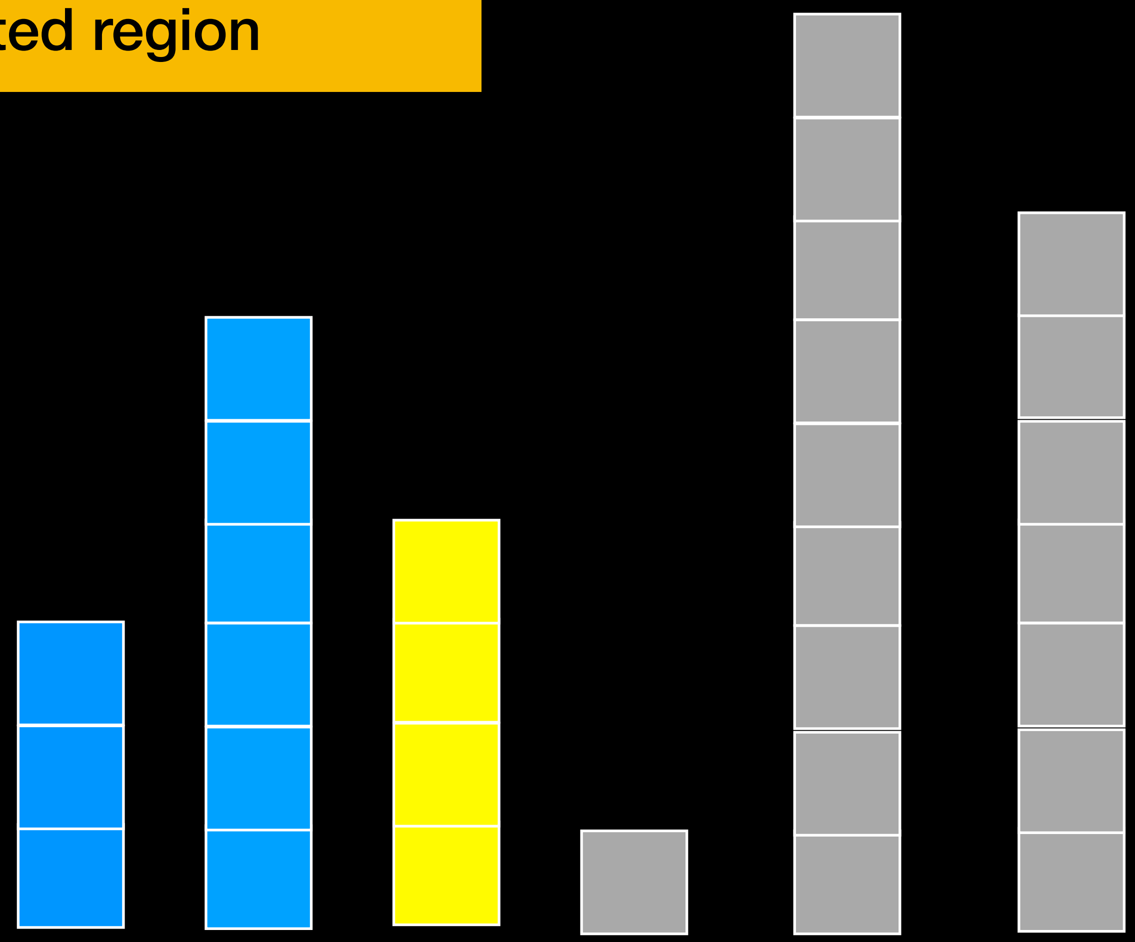


Insertion Sort

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

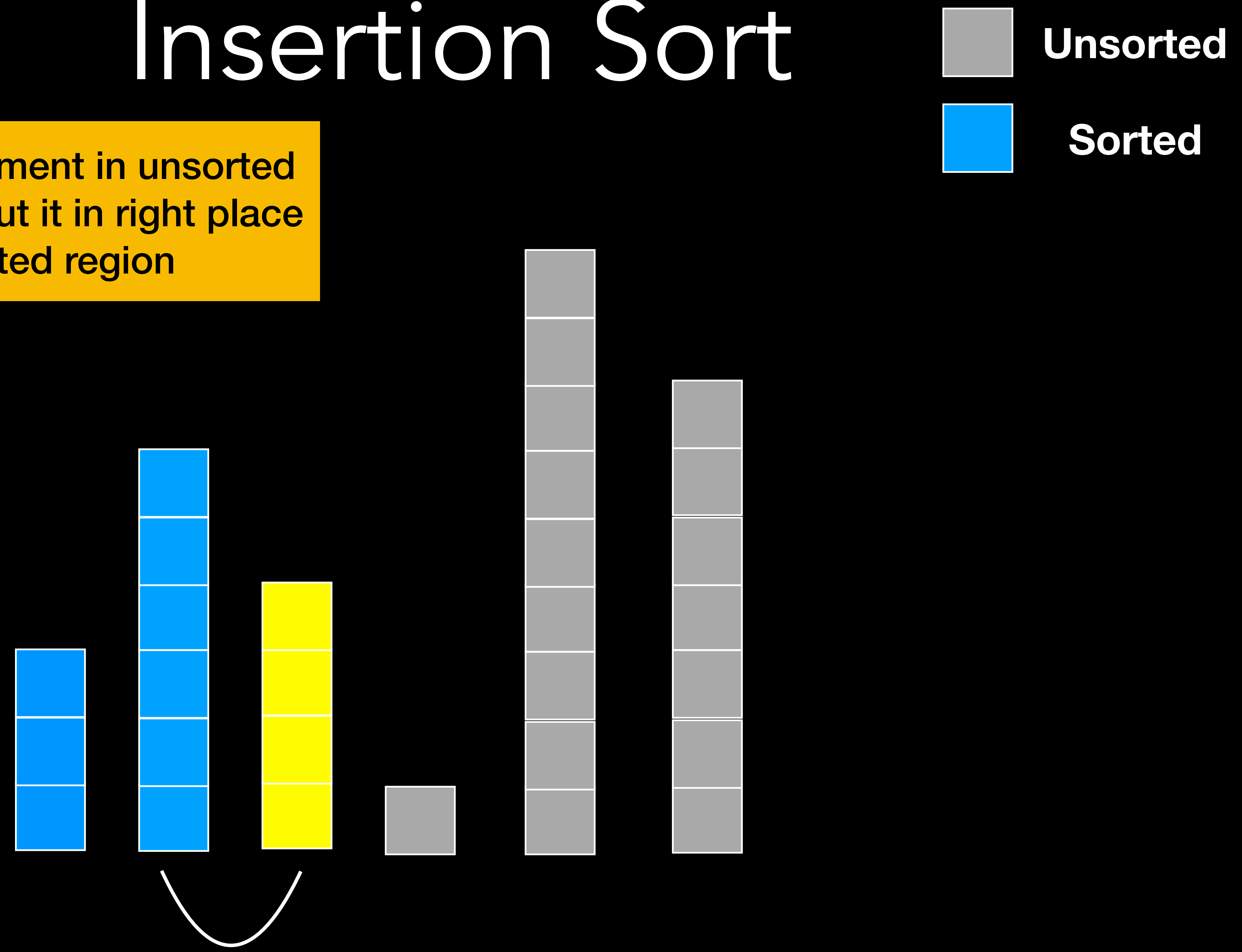
Unsorted

Sorted

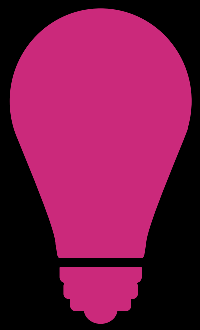


Insertion Sort

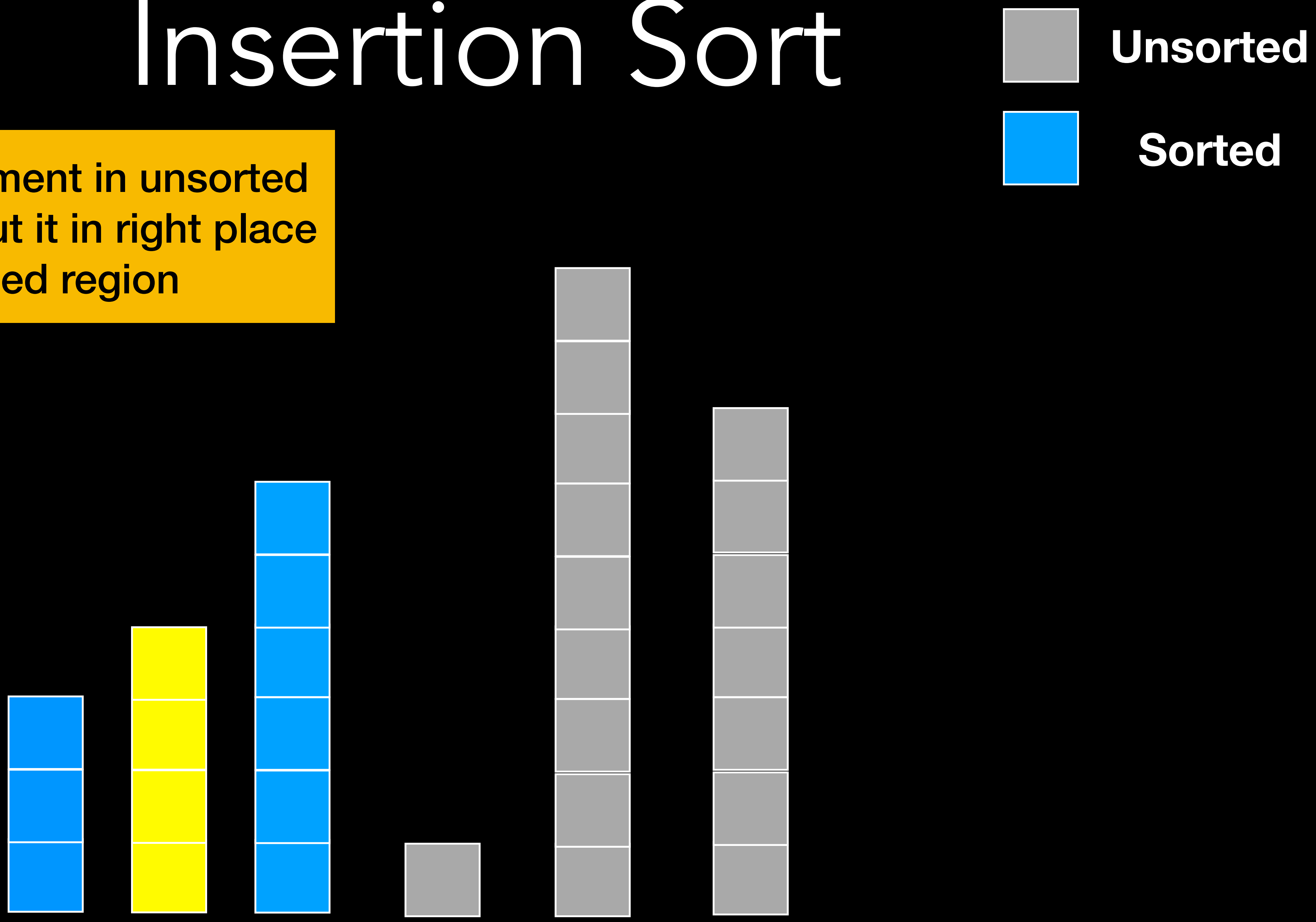
Pick first element in unsorted region and put it in right place in sorted region



Insertion Sort



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

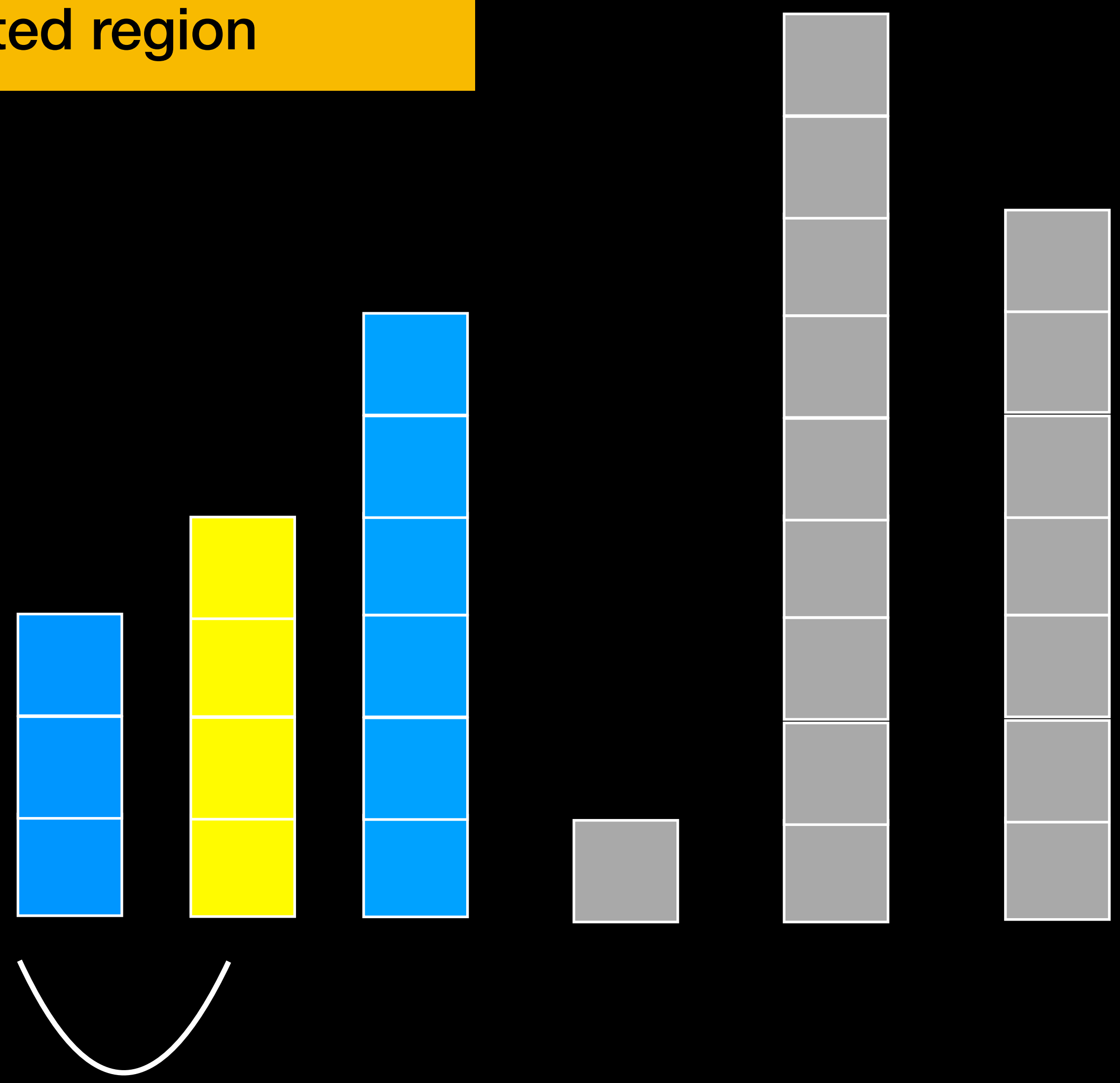


Insertion Sort

Unsorted

Sorted

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

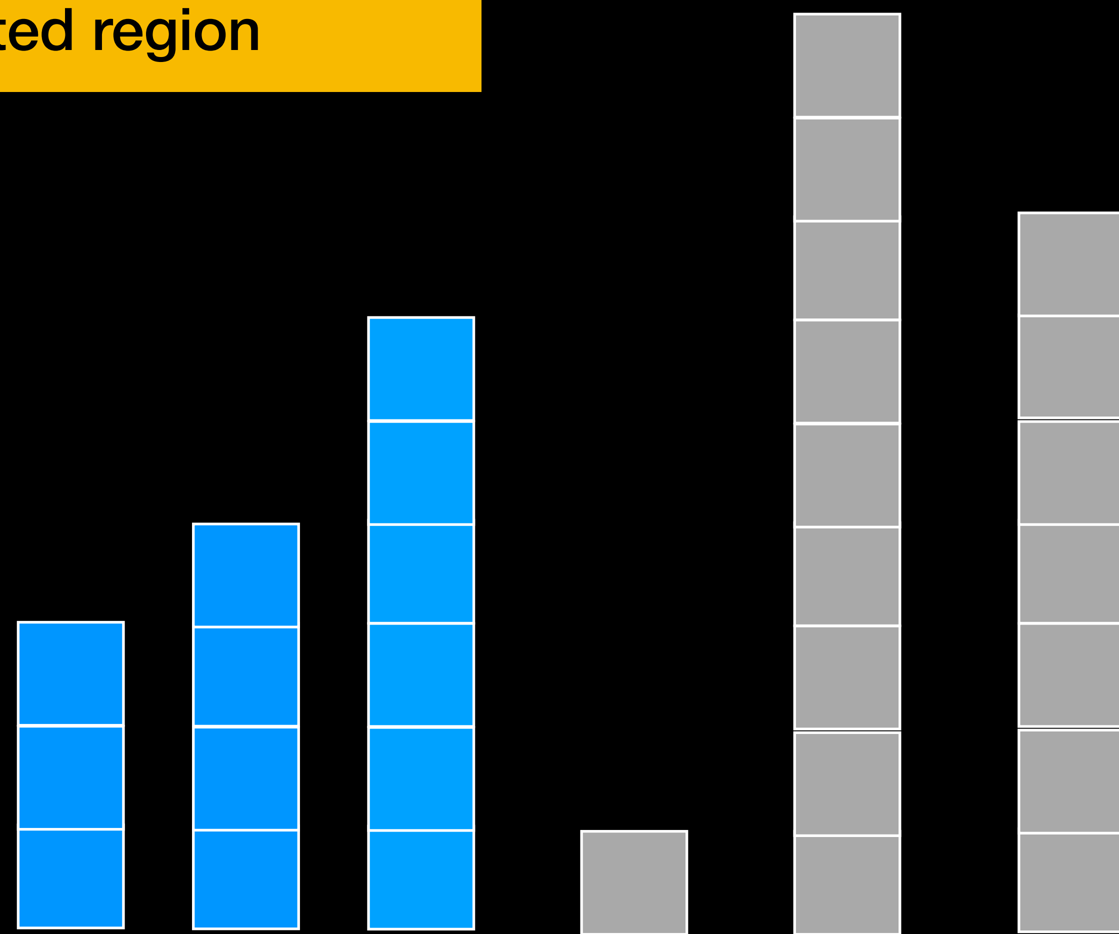


Insertion Sort

Unsorted

Sorted

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

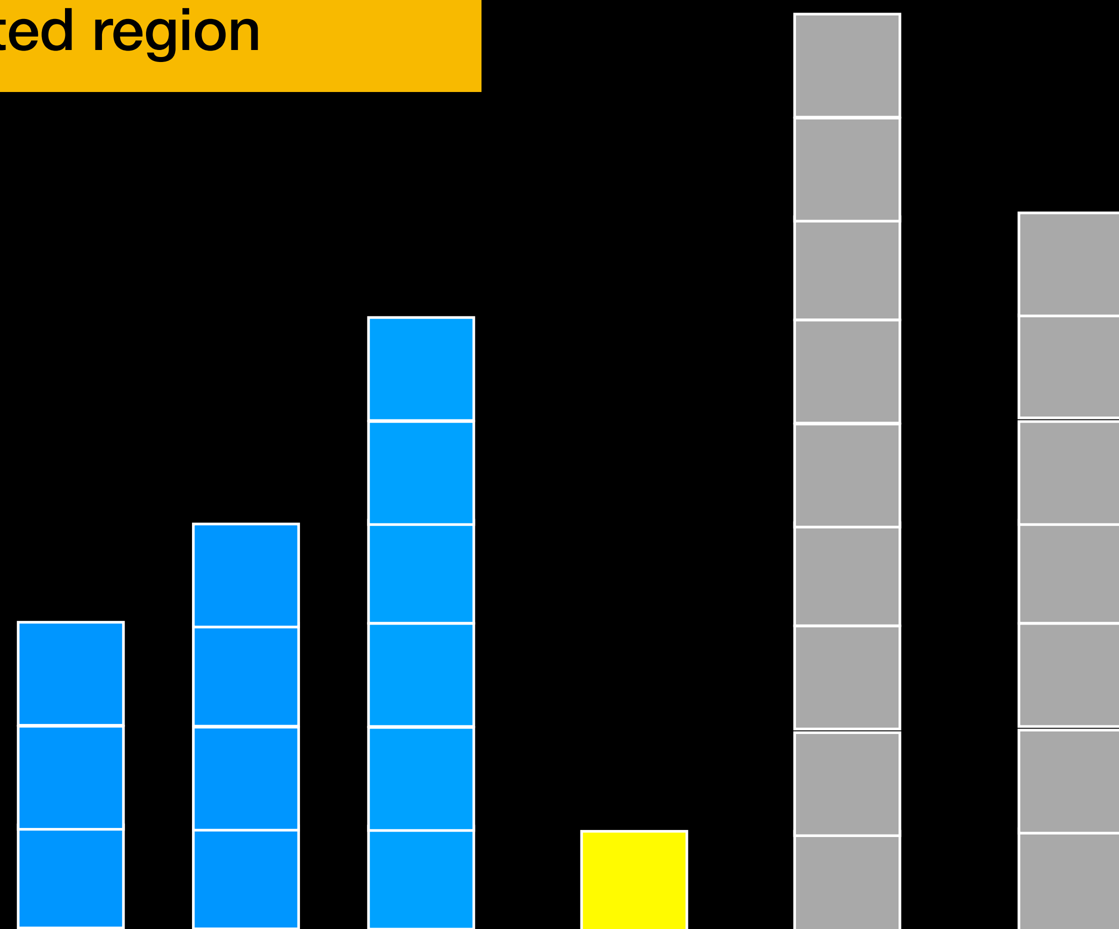


Insertion Sort

Unsorted

Sorted

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

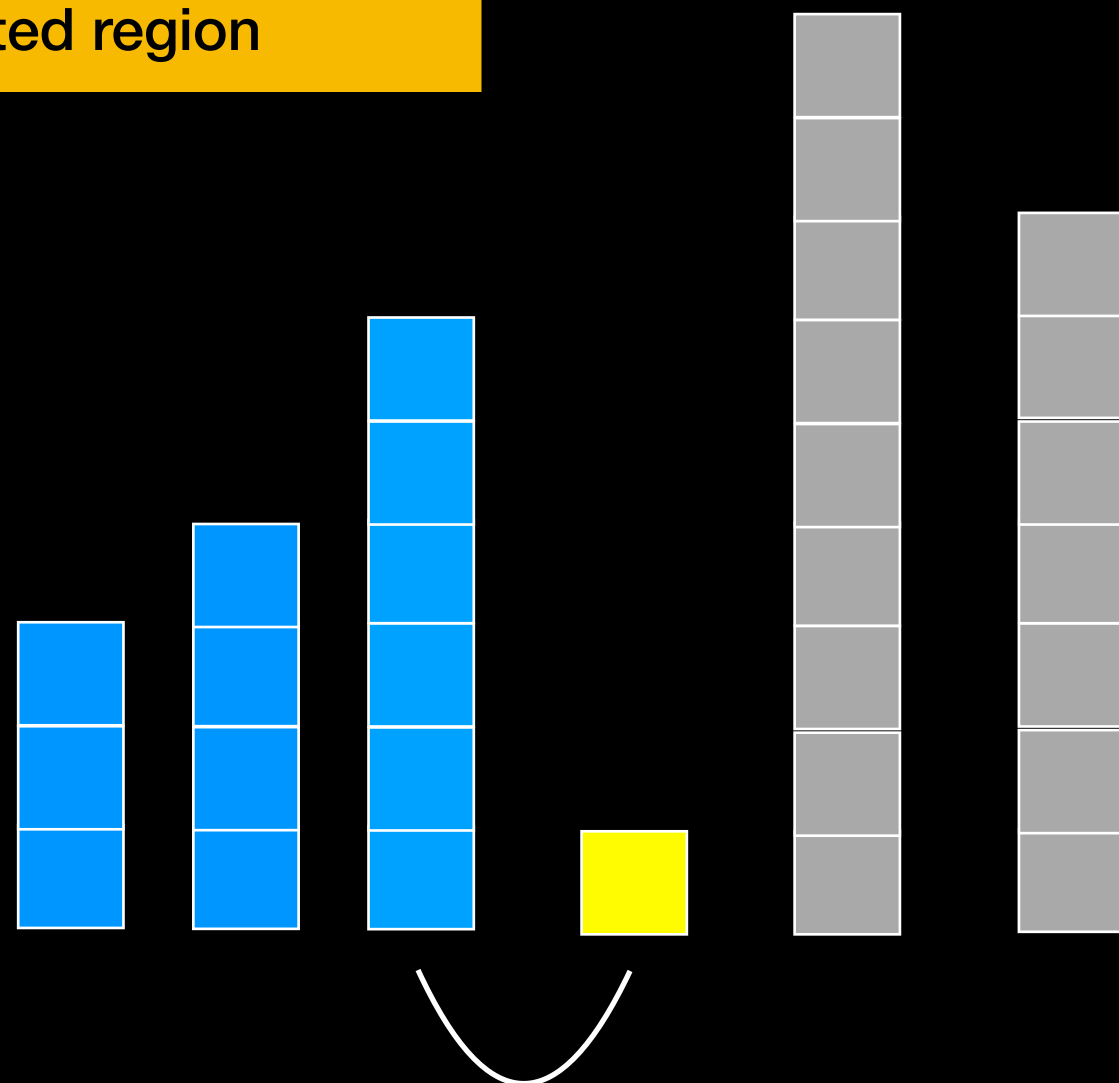


Insertion Sort

Unsorted

Sorted

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

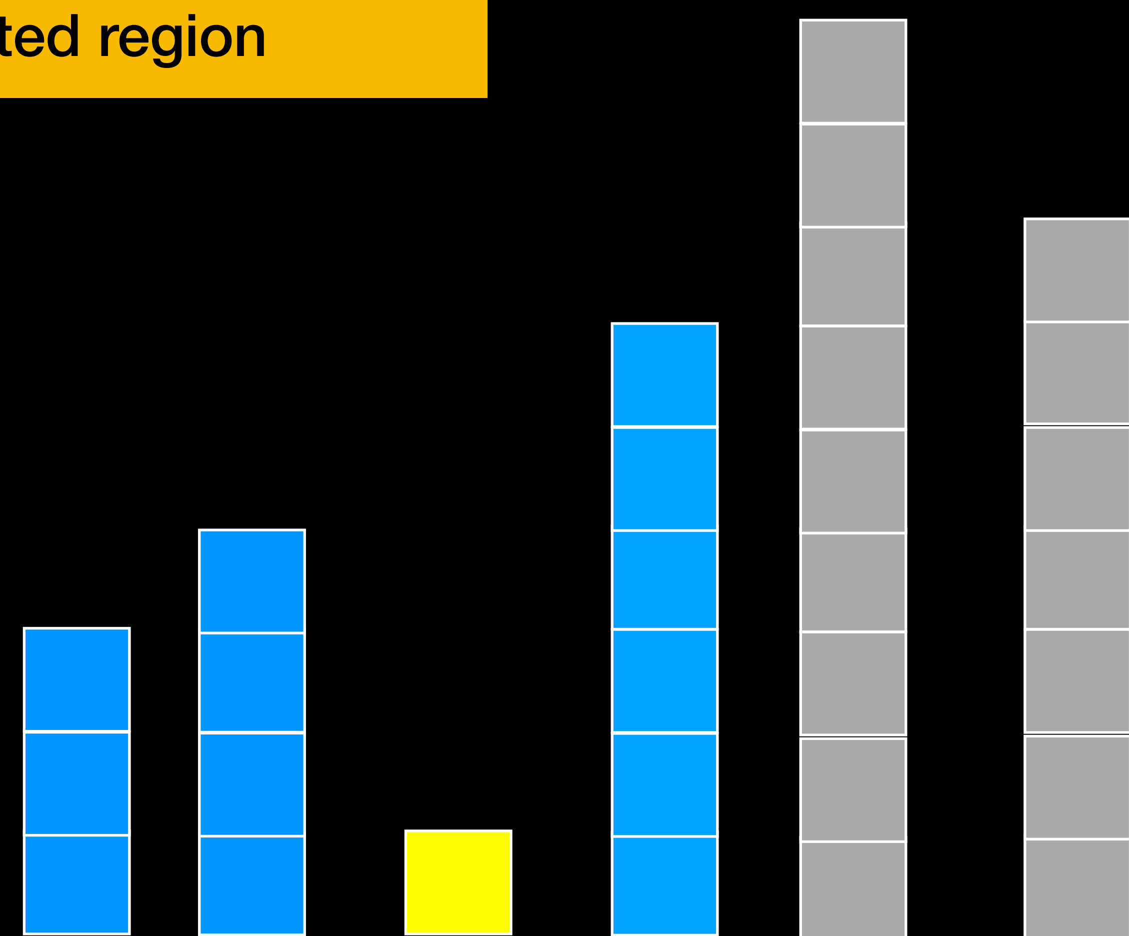


Insertion Sort

Unsorted

Sorted

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

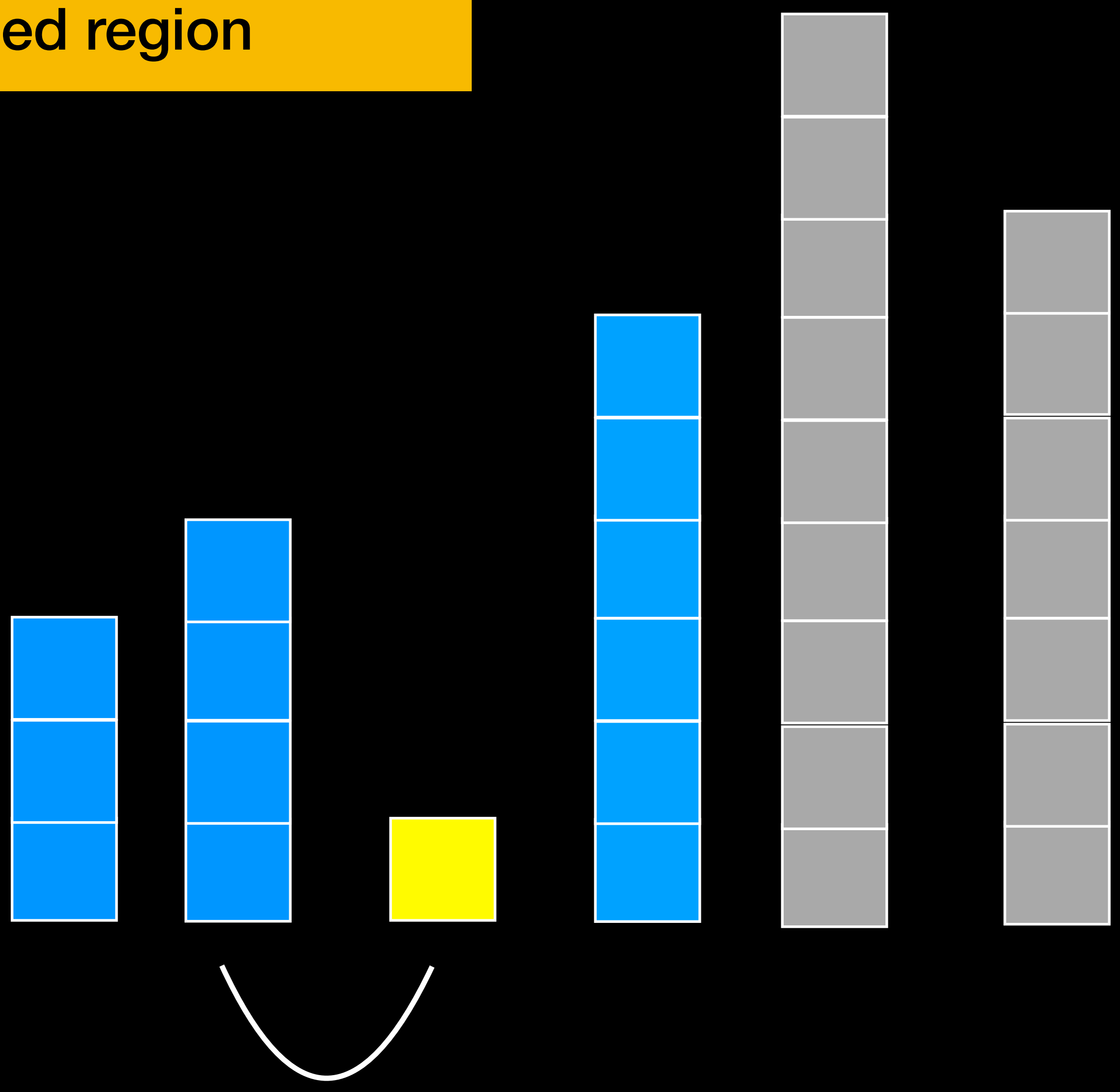


Insertion Sort

Unsorted

Sorted

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

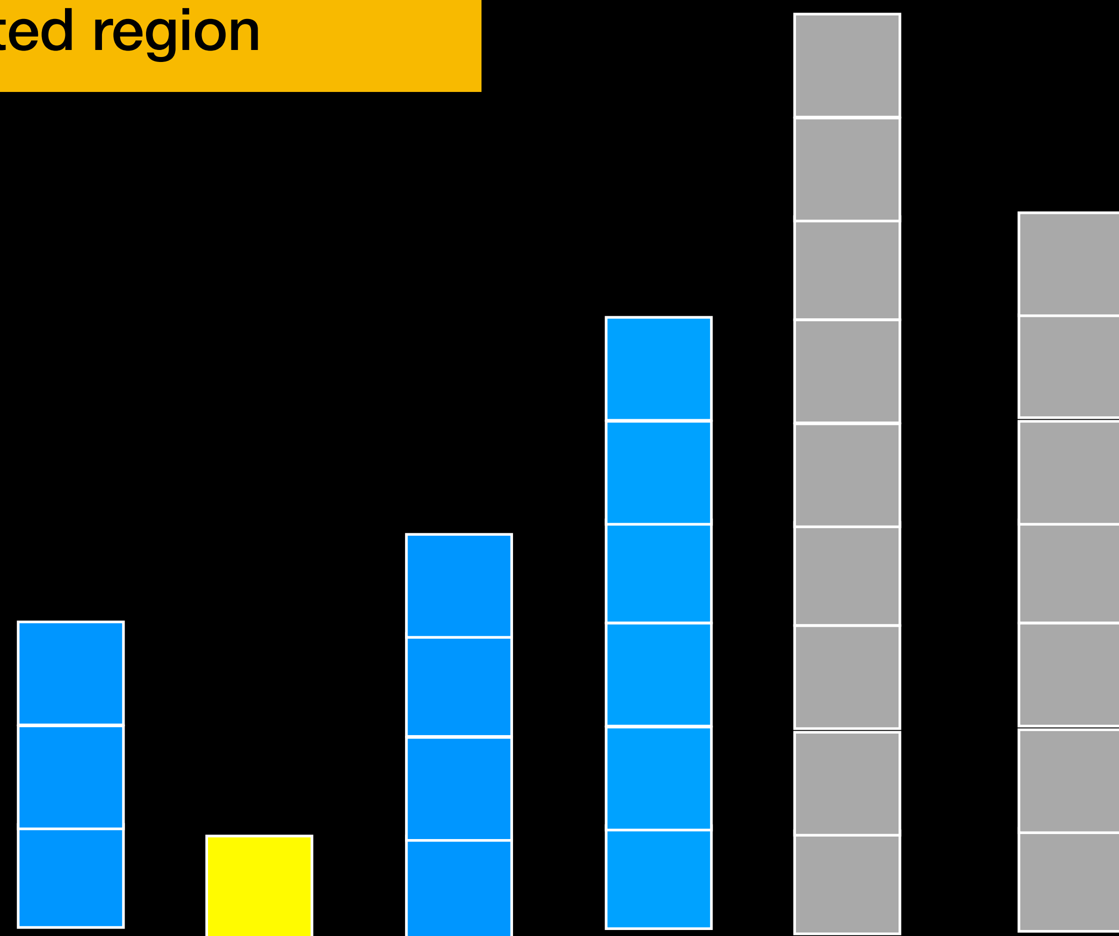


Insertion Sort

Unsorted

Sorted

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

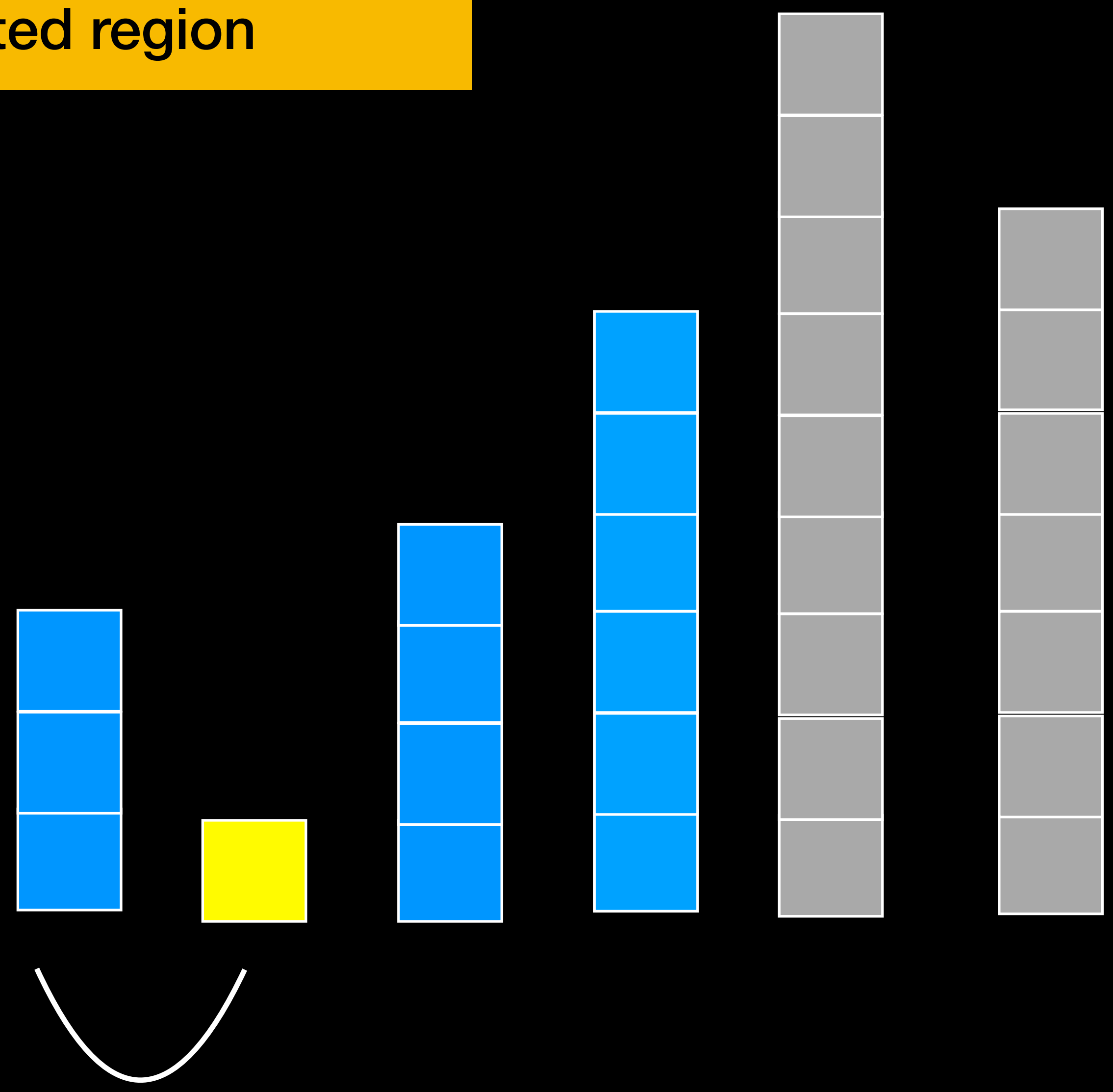


Insertion Sort

Unsorted

Sorted

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

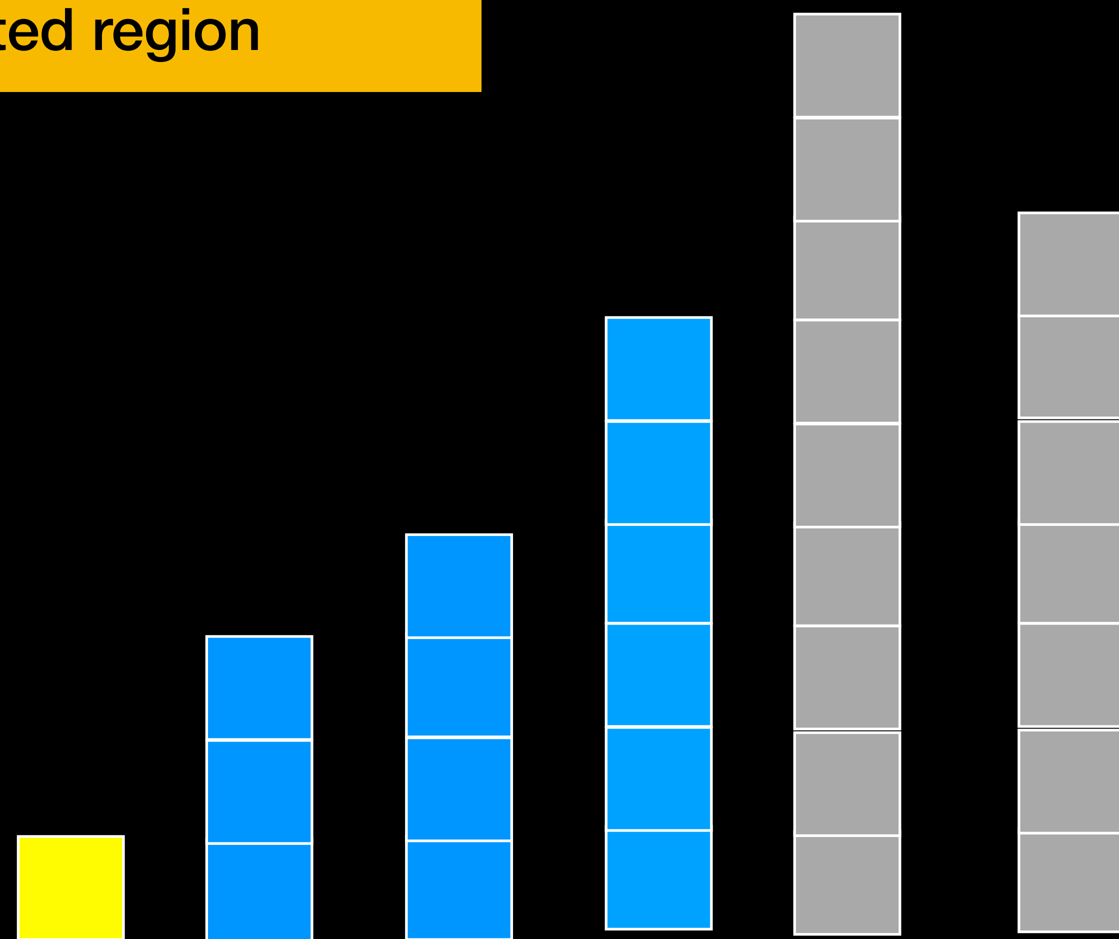


Insertion Sort

Unsorted

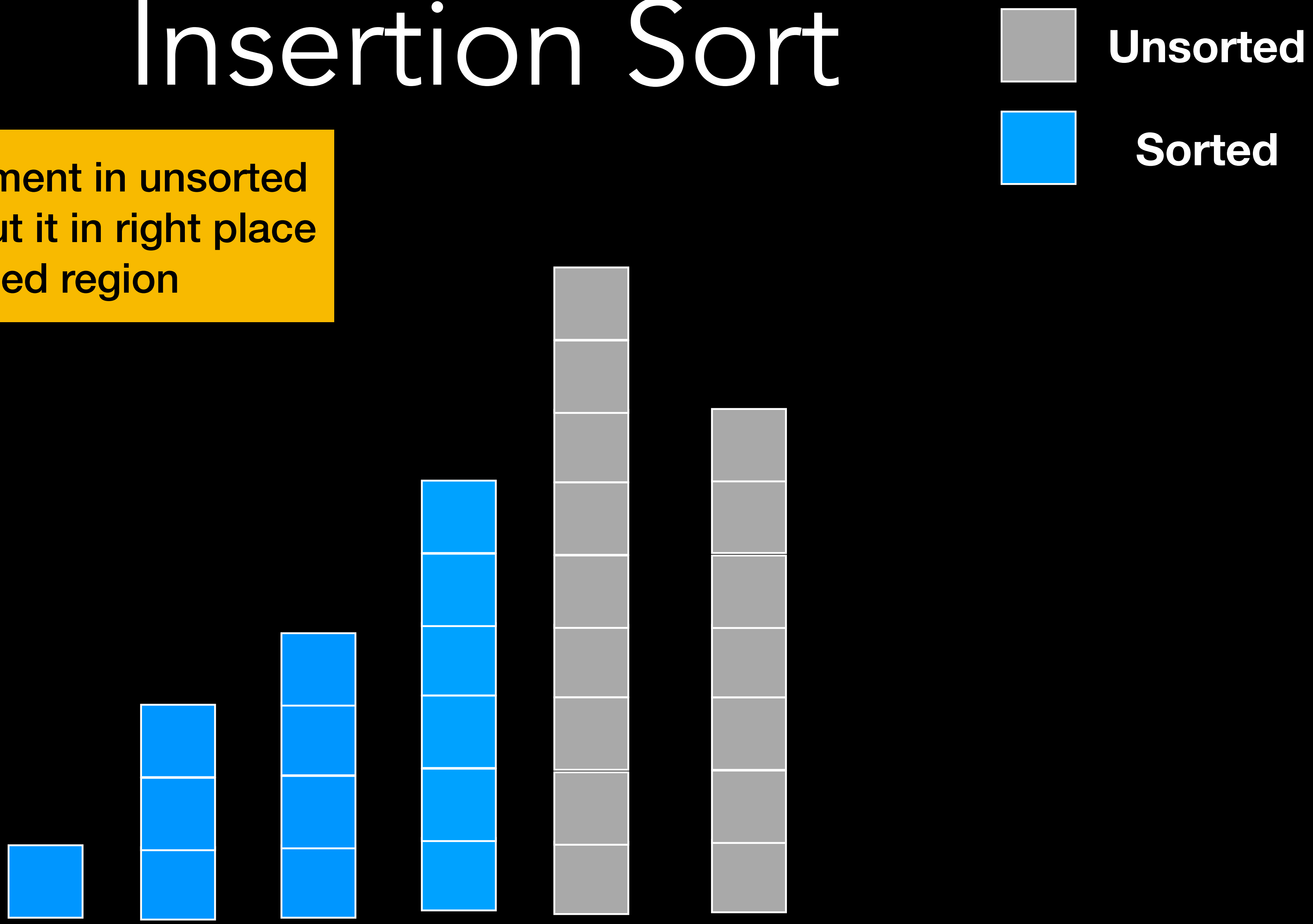
Sorted

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

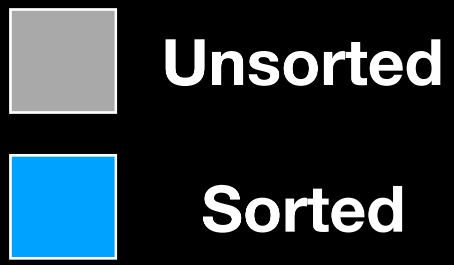


Insertion Sort

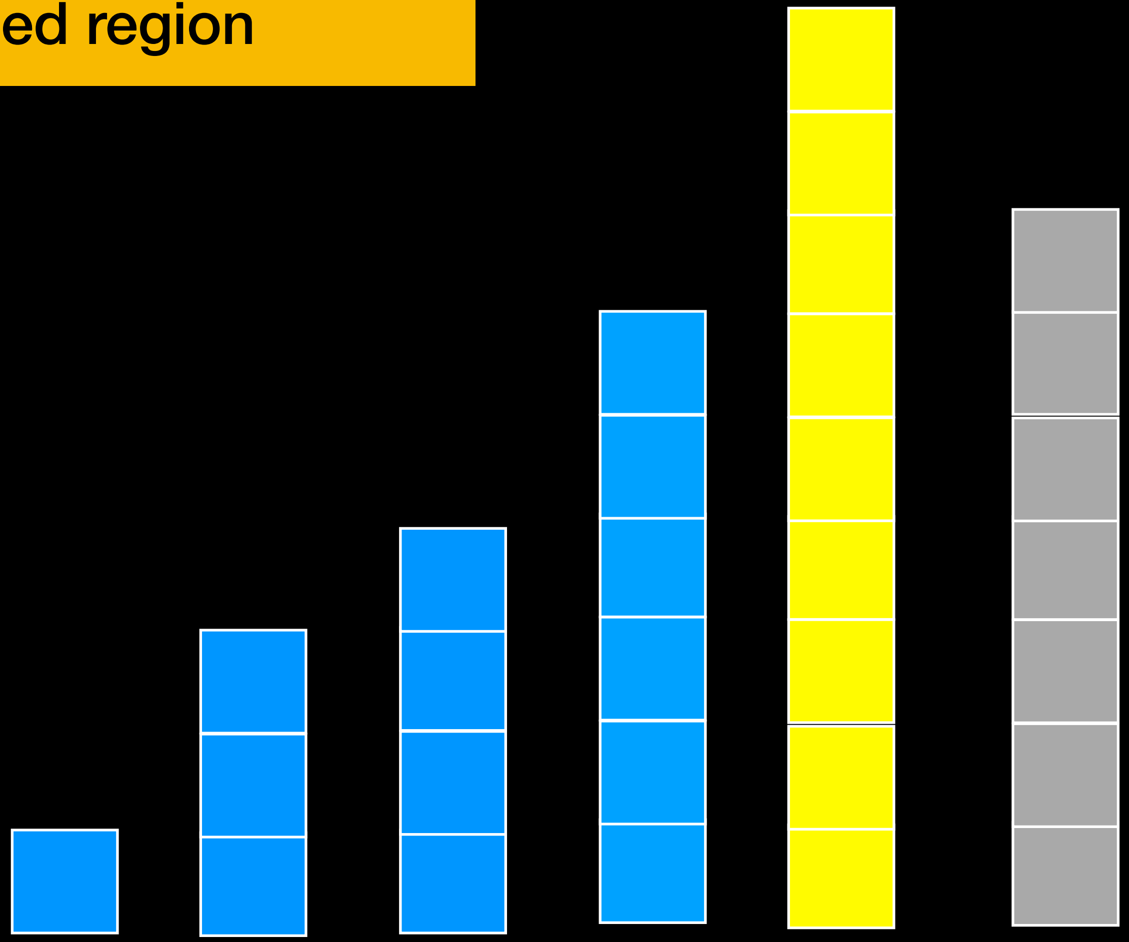
Pick first element in unsorted region and put it in right place in sorted region



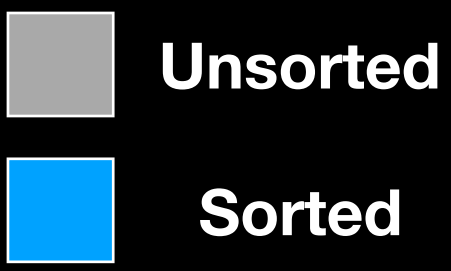
Insertion Sort



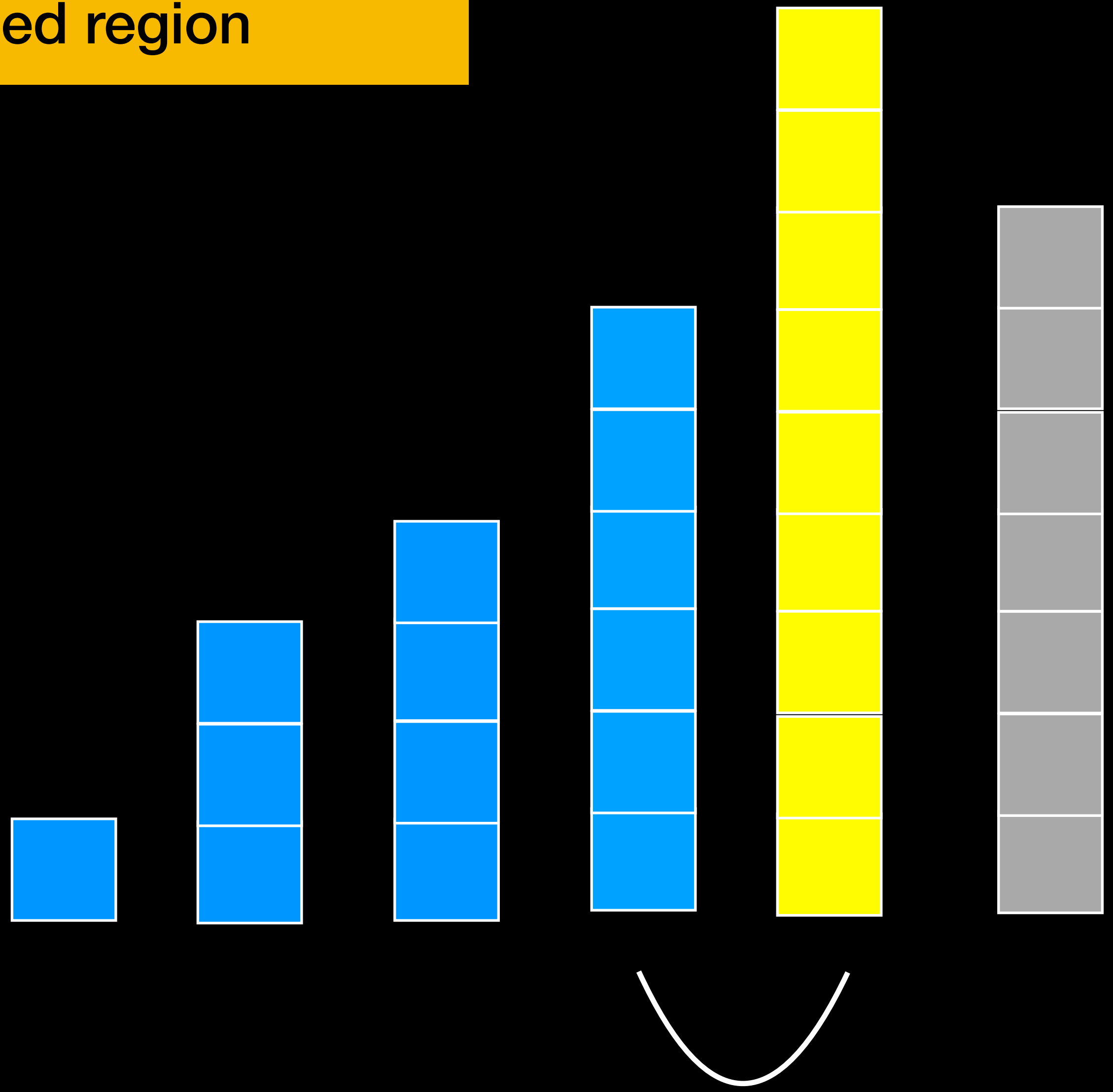
Pick first element in unsorted region and put it in right place in sorted region



Insertion Sort



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

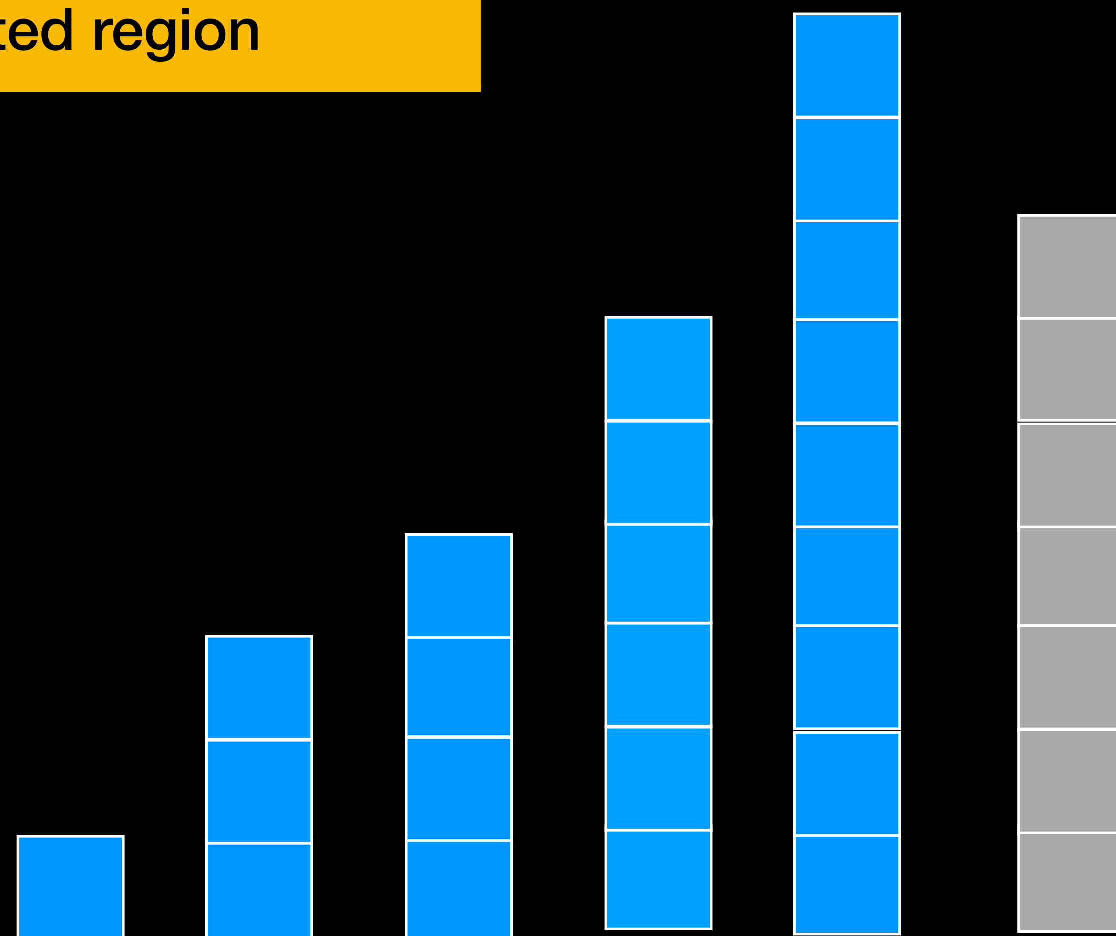


Insertion Sort

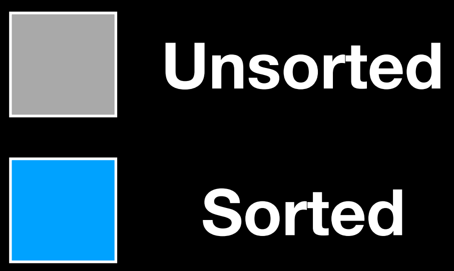
Unsorted

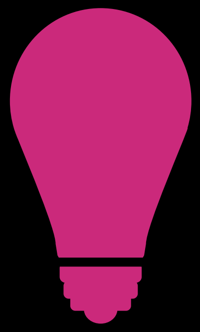
Sorted

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

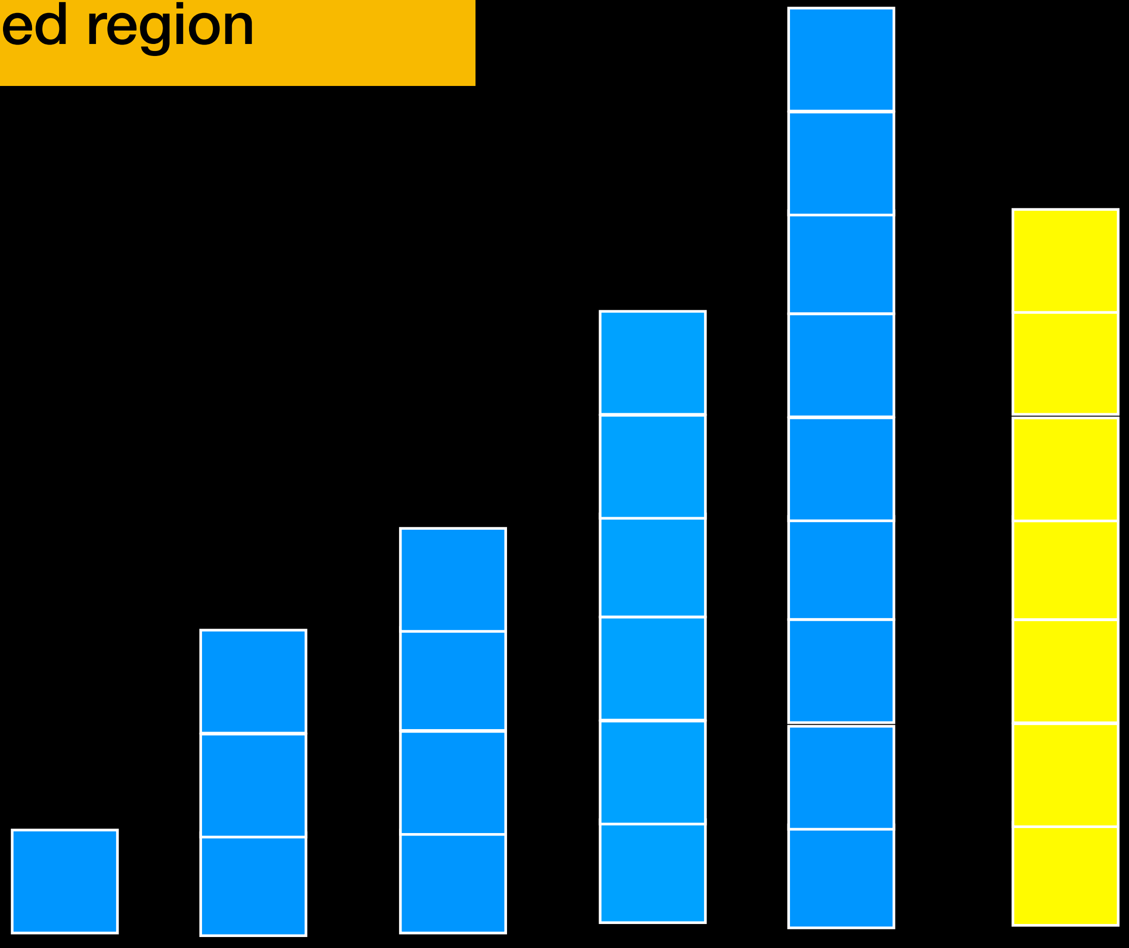


Insertion Sort

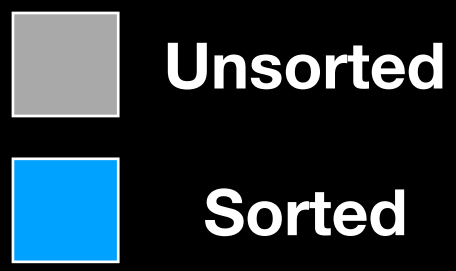




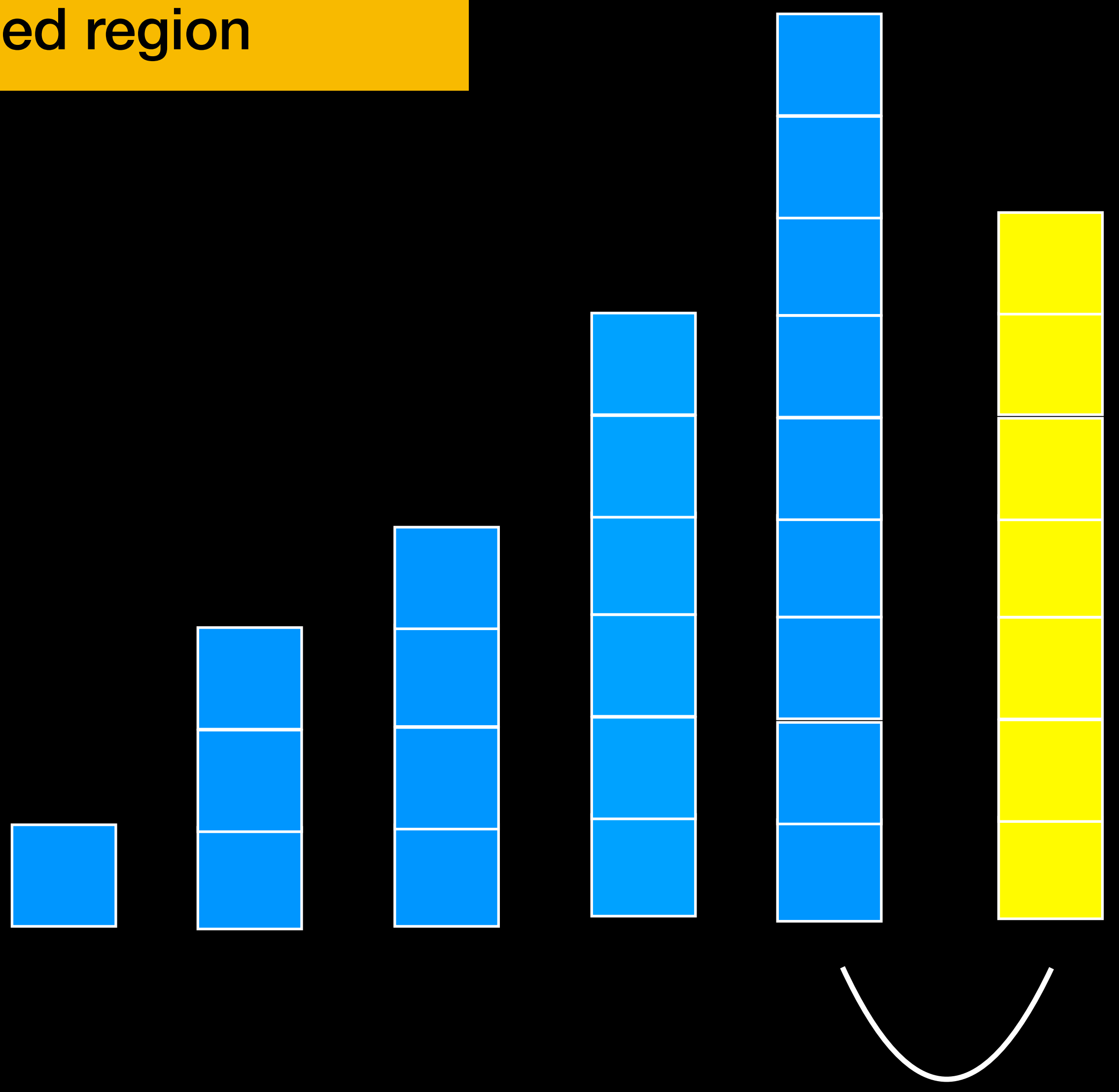
Pick first element in unsorted region and put it in right place in sorted region



Insertion Sort



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

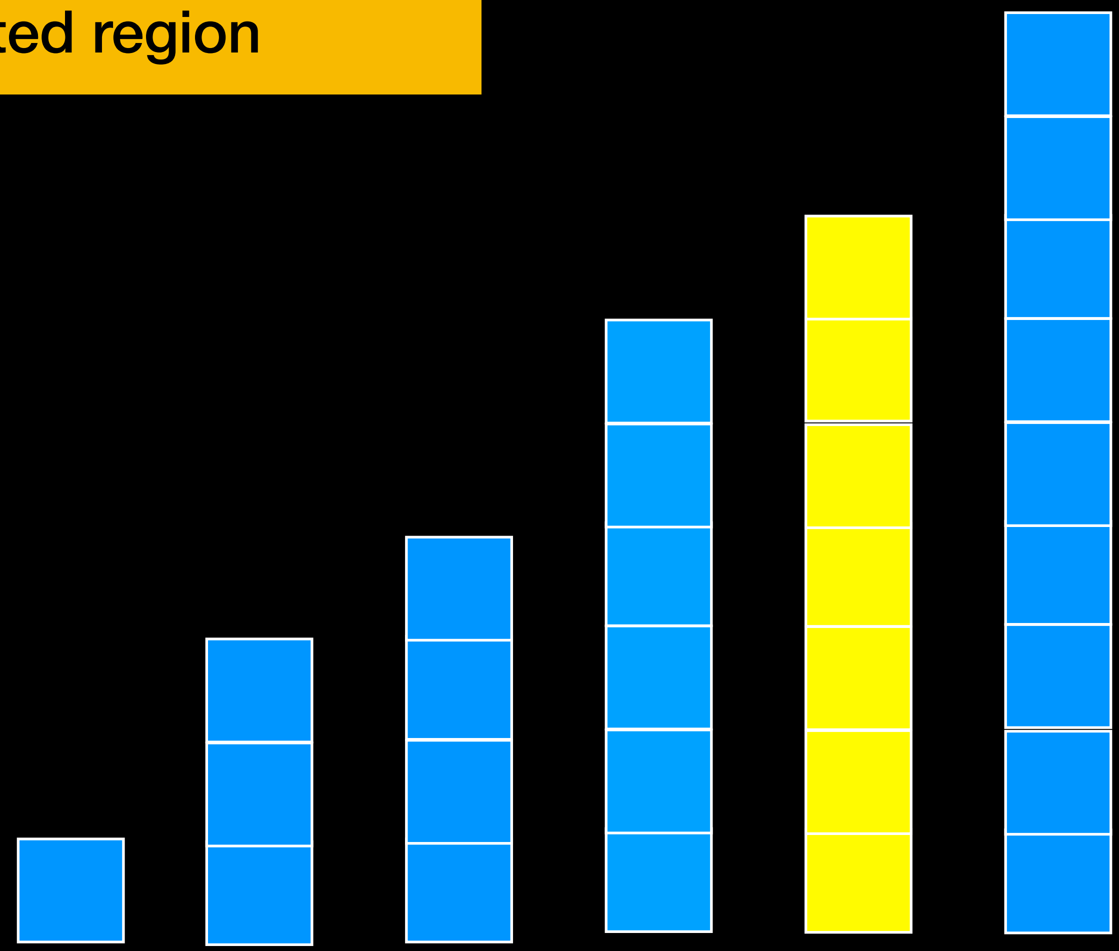


Insertion Sort

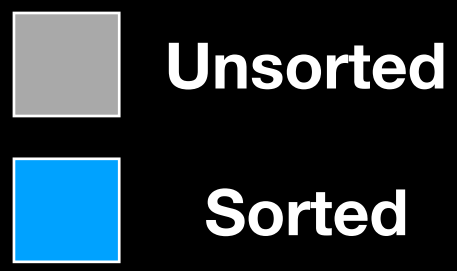
Unsorted

Sorted

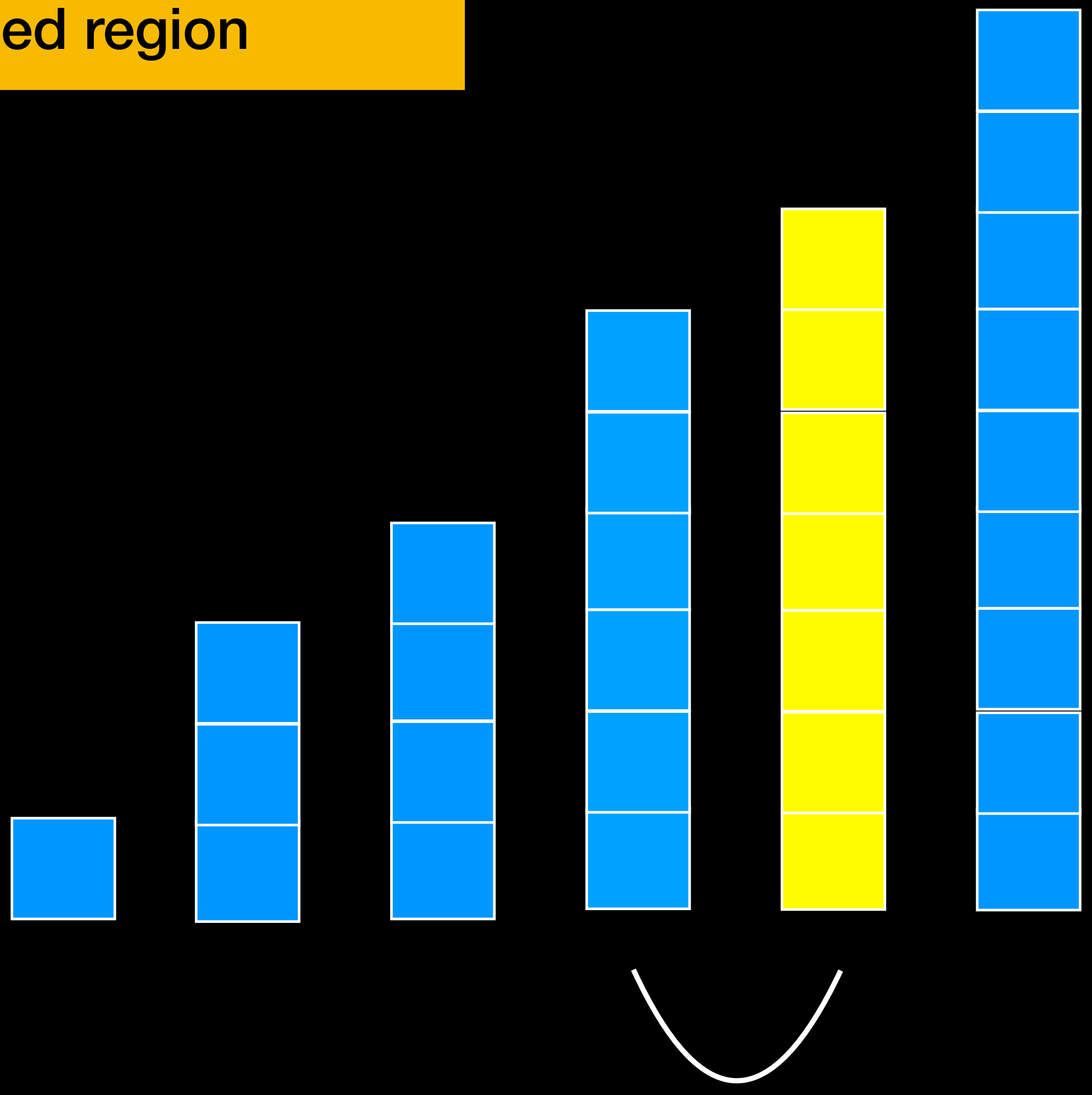
Pick first element in unsorted region and put it in right place in sorted region



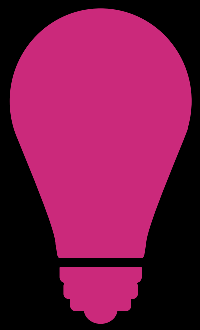
Insertion Sort



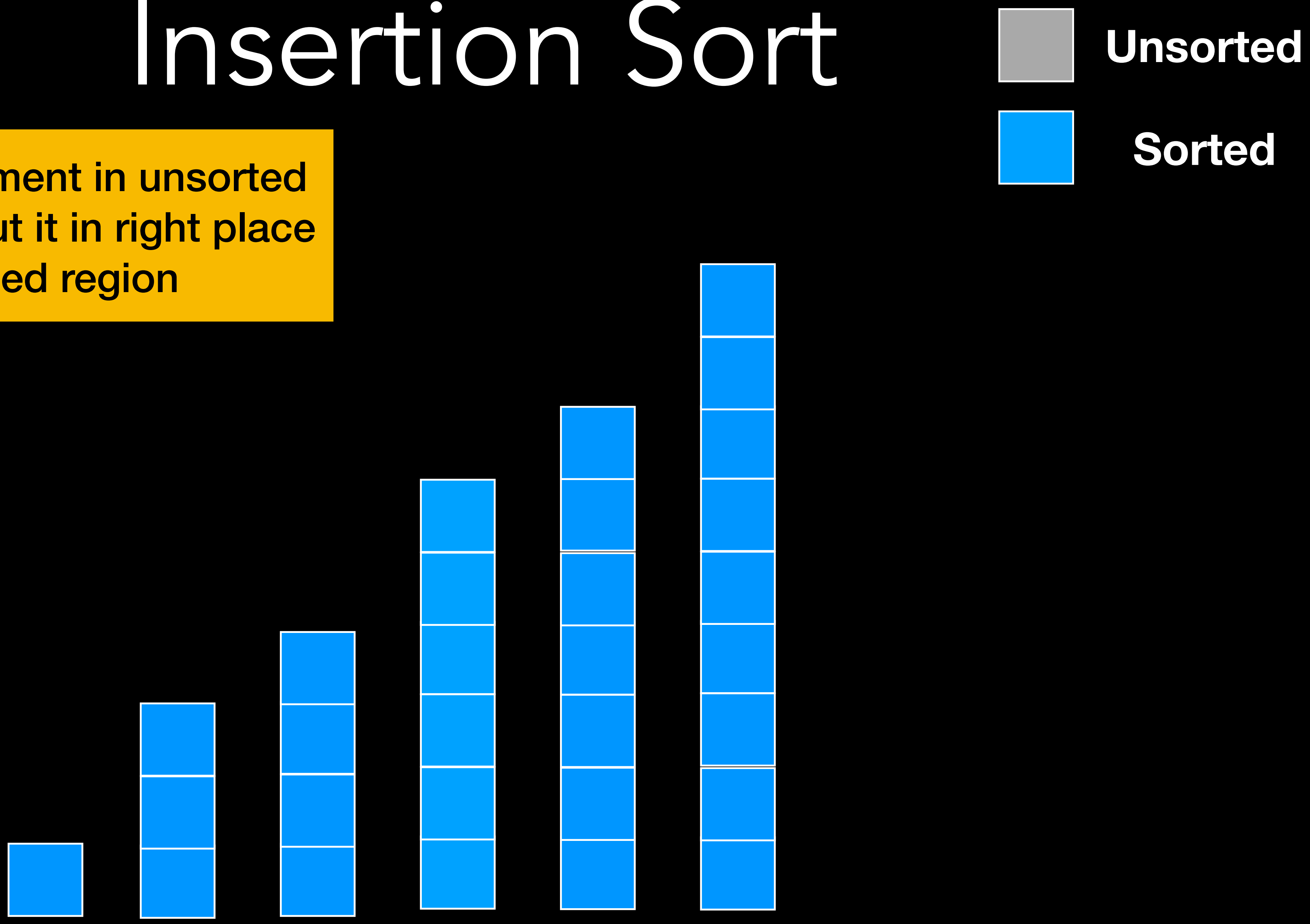
Pick first element in unsorted region and put it in right place in sorted region



Insertion Sort



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** swaps

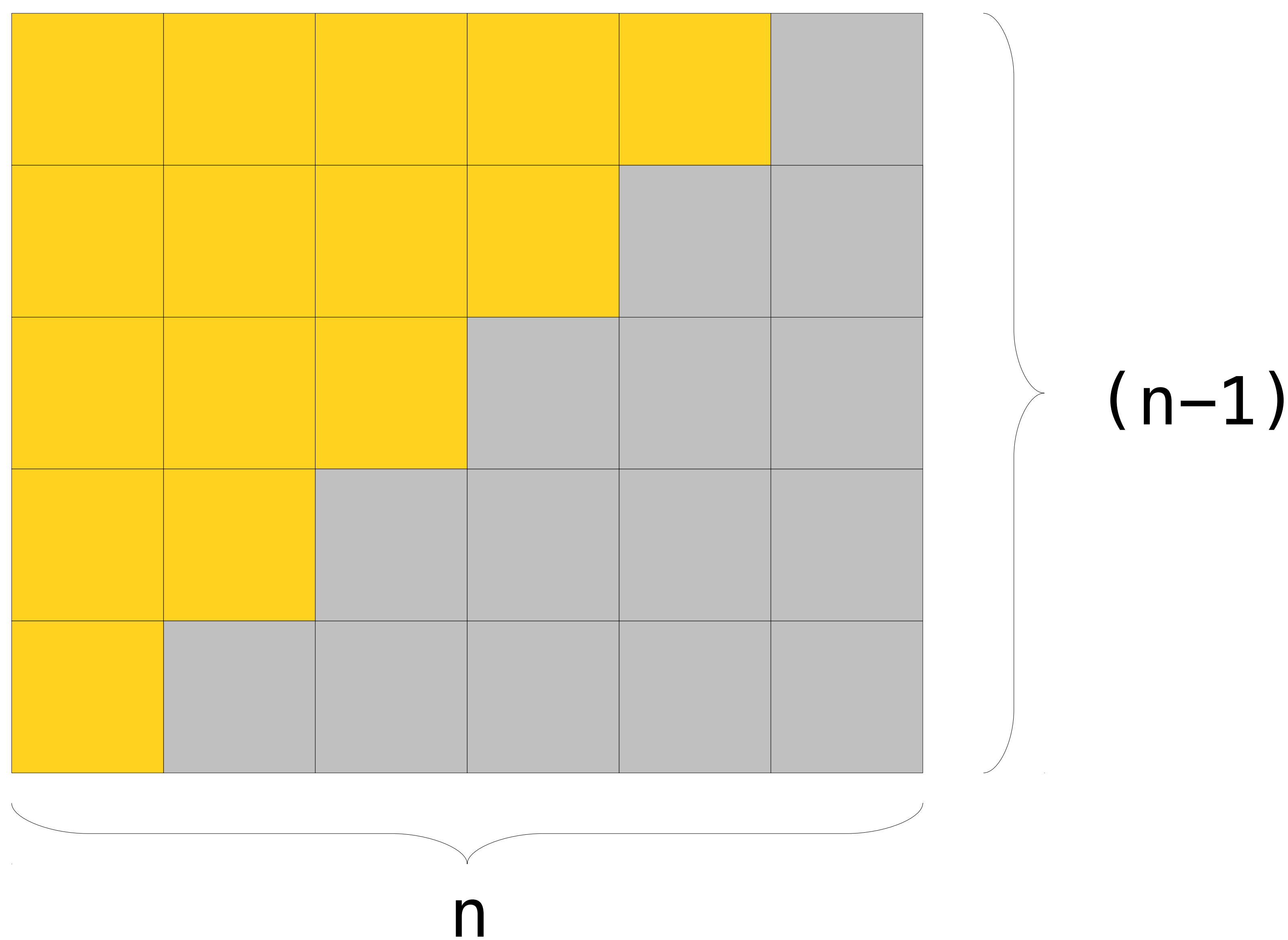
Second pass: at most **2** comparisons and at most **2** swaps

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

. . .

Total work: **1 + 2 + 3 + . . . + (n-1)**

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



Insertion Sort Analysis

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

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

$T(n) = n^2-n = O(\ \mathbf{n^2})$

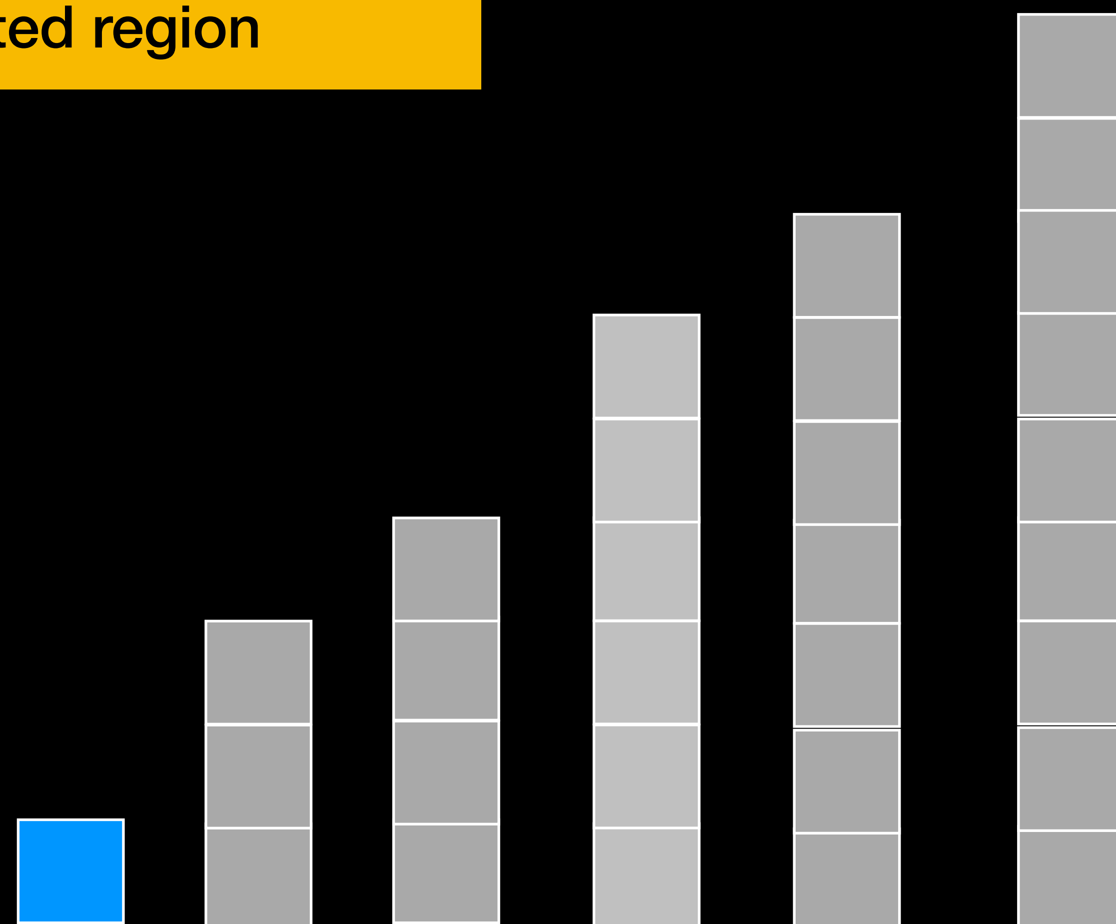
Insertion Sort run time is **$O(\ n^2)$**

Insertion Sort

Unsorted

Sorted

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

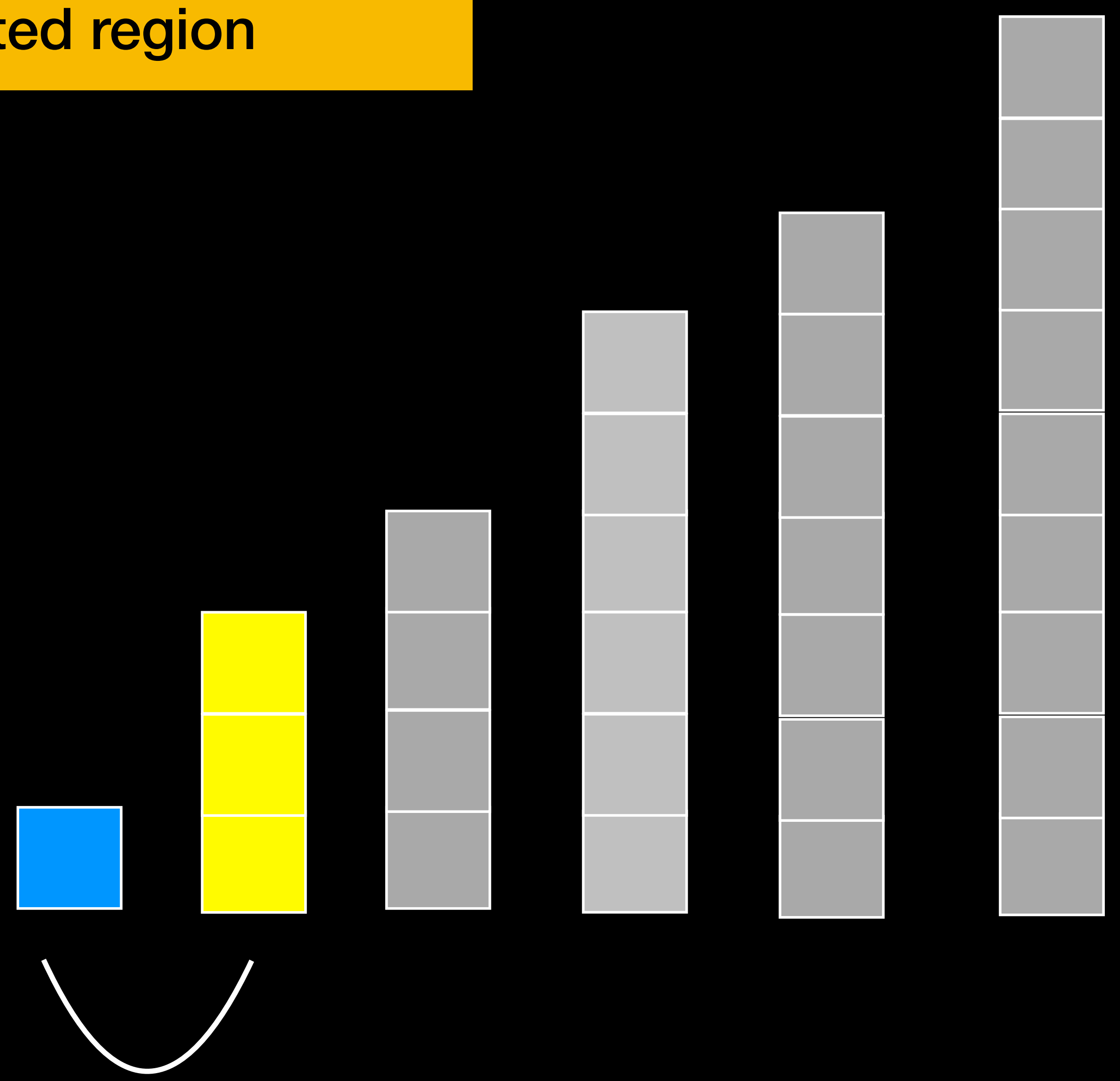


Insertion Sort

Unsorted

Sorted

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

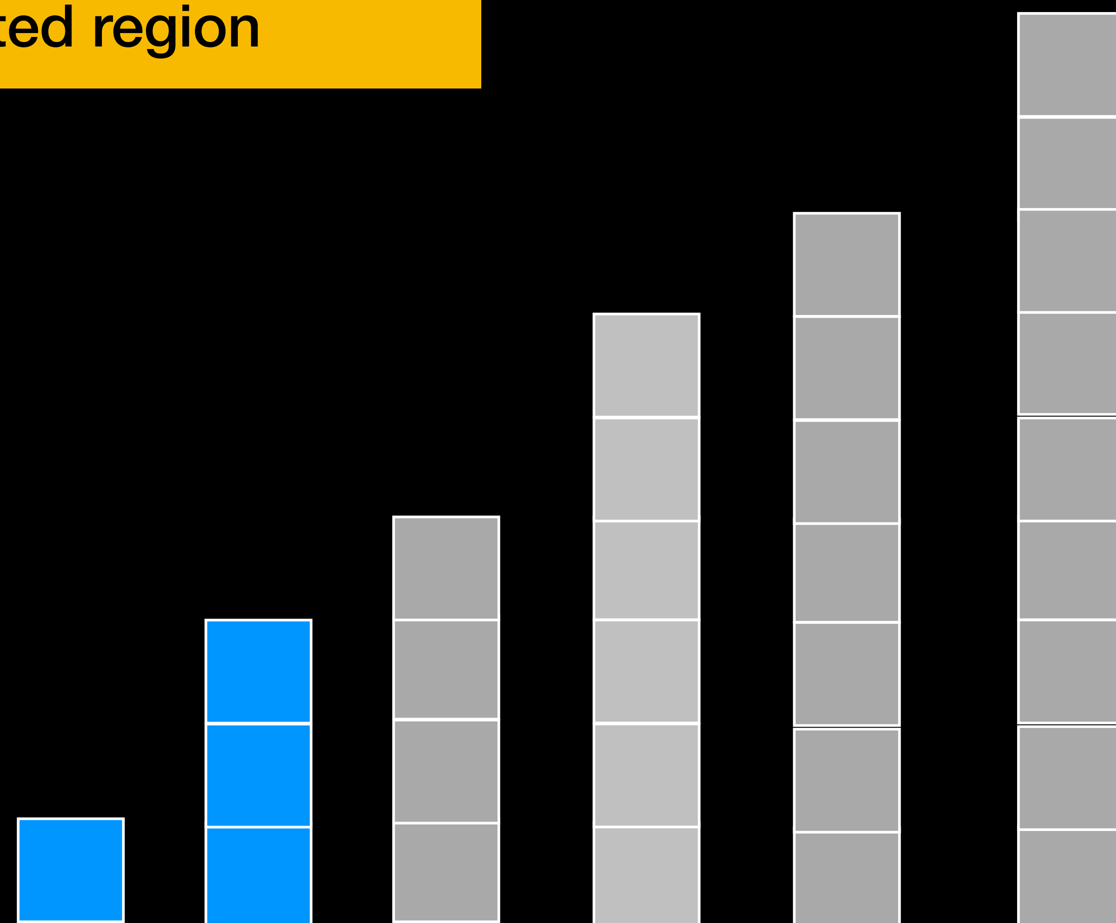


Insertion Sort

Unsorted

Sorted

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

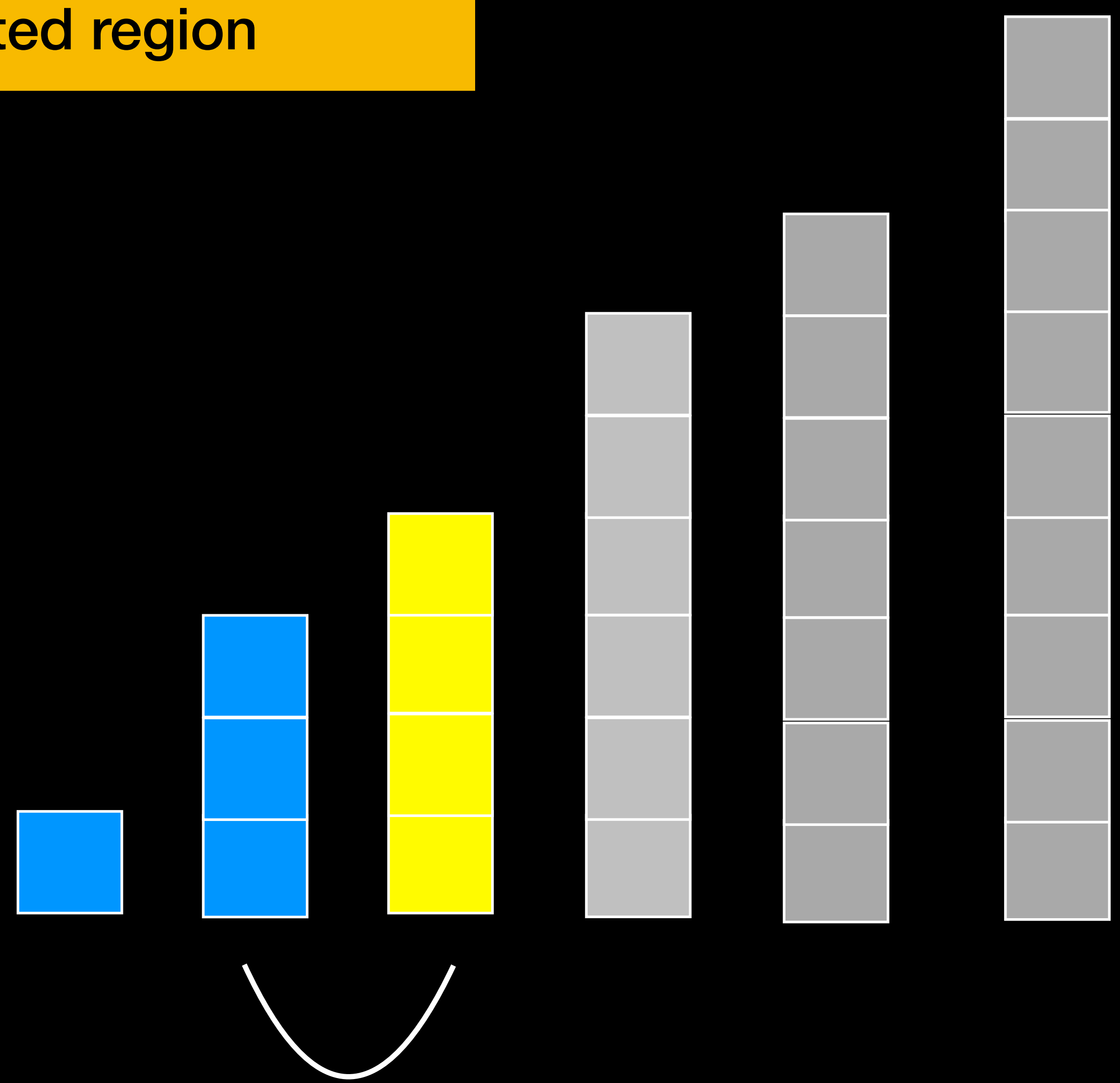


Insertion Sort

Unsorted

Sorted

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

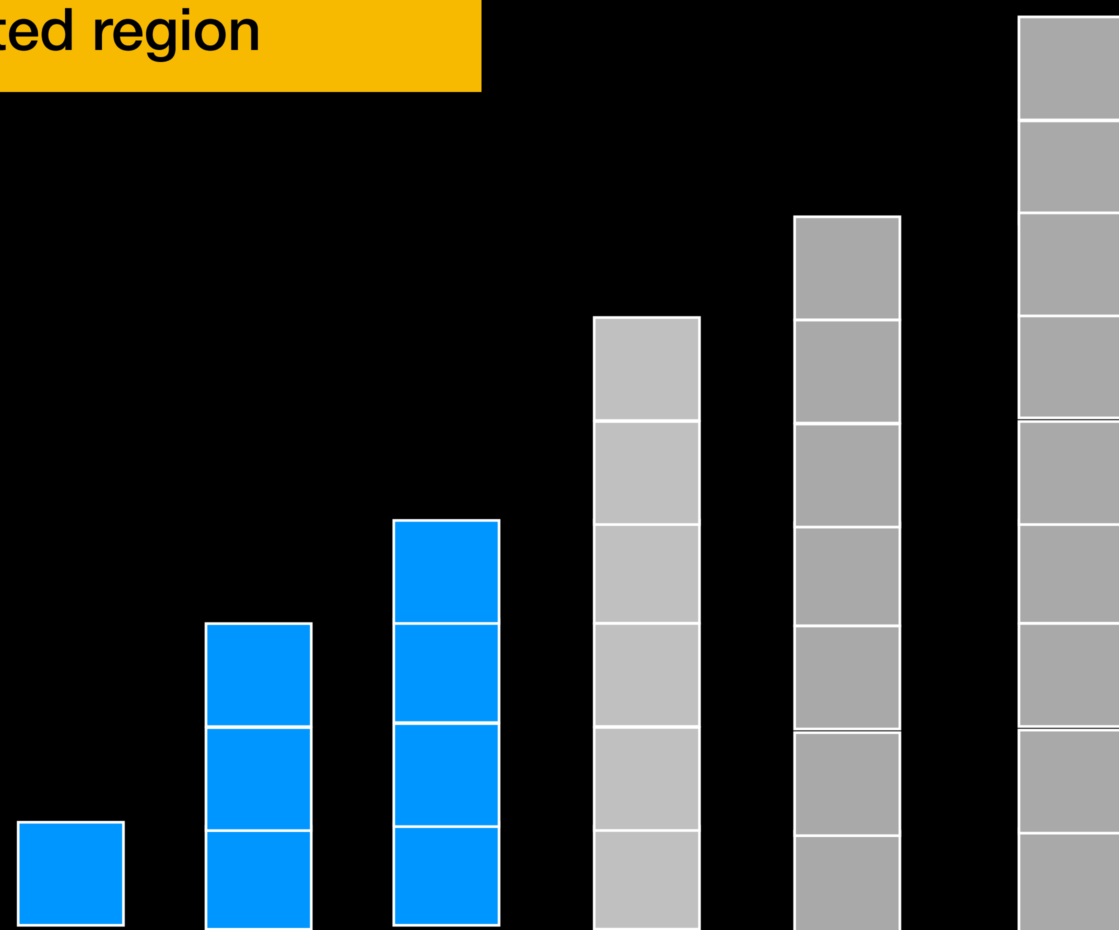


Insertion Sort

Unsorted

Sorted

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

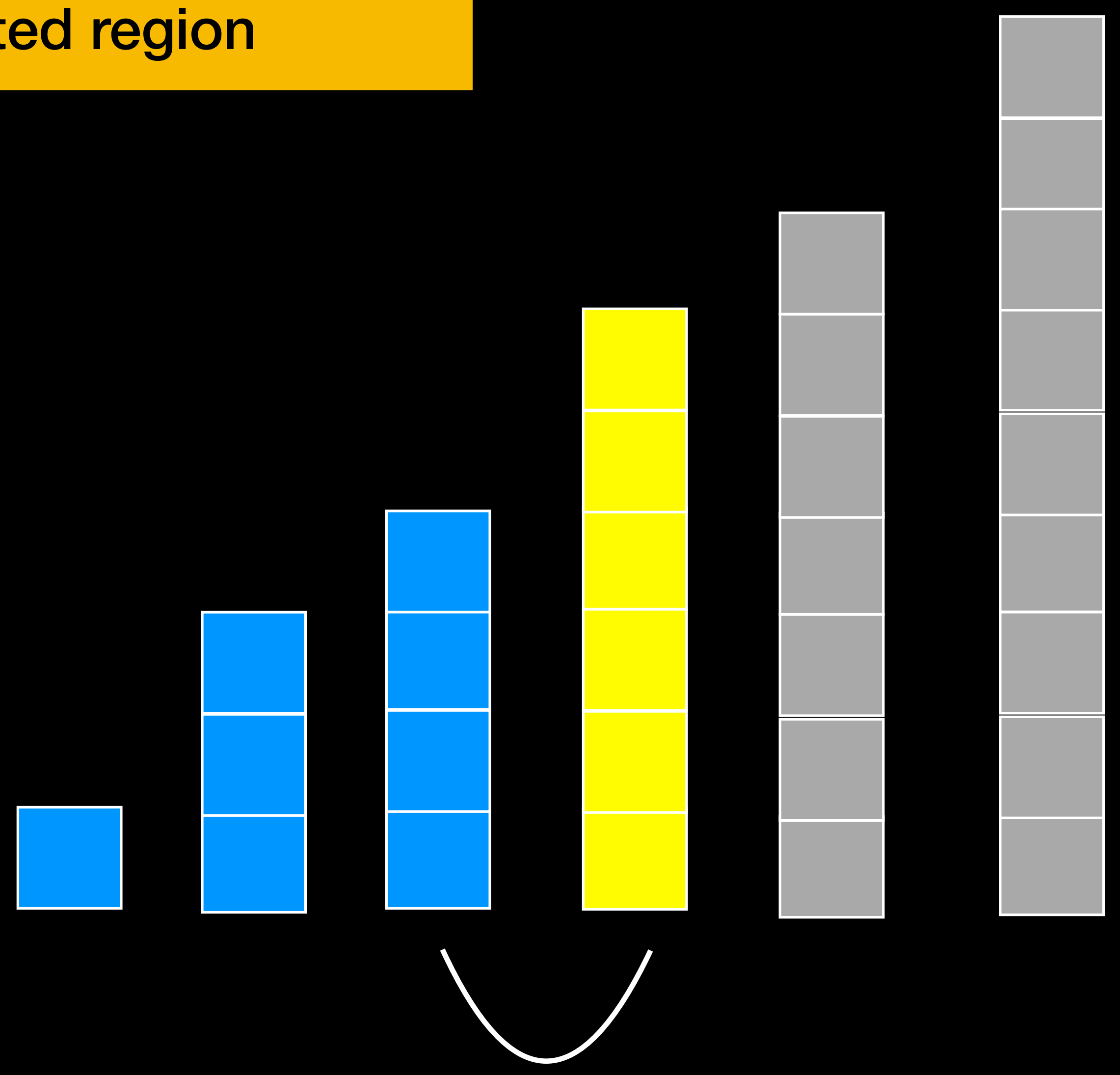


Insertion Sort

Unsorted

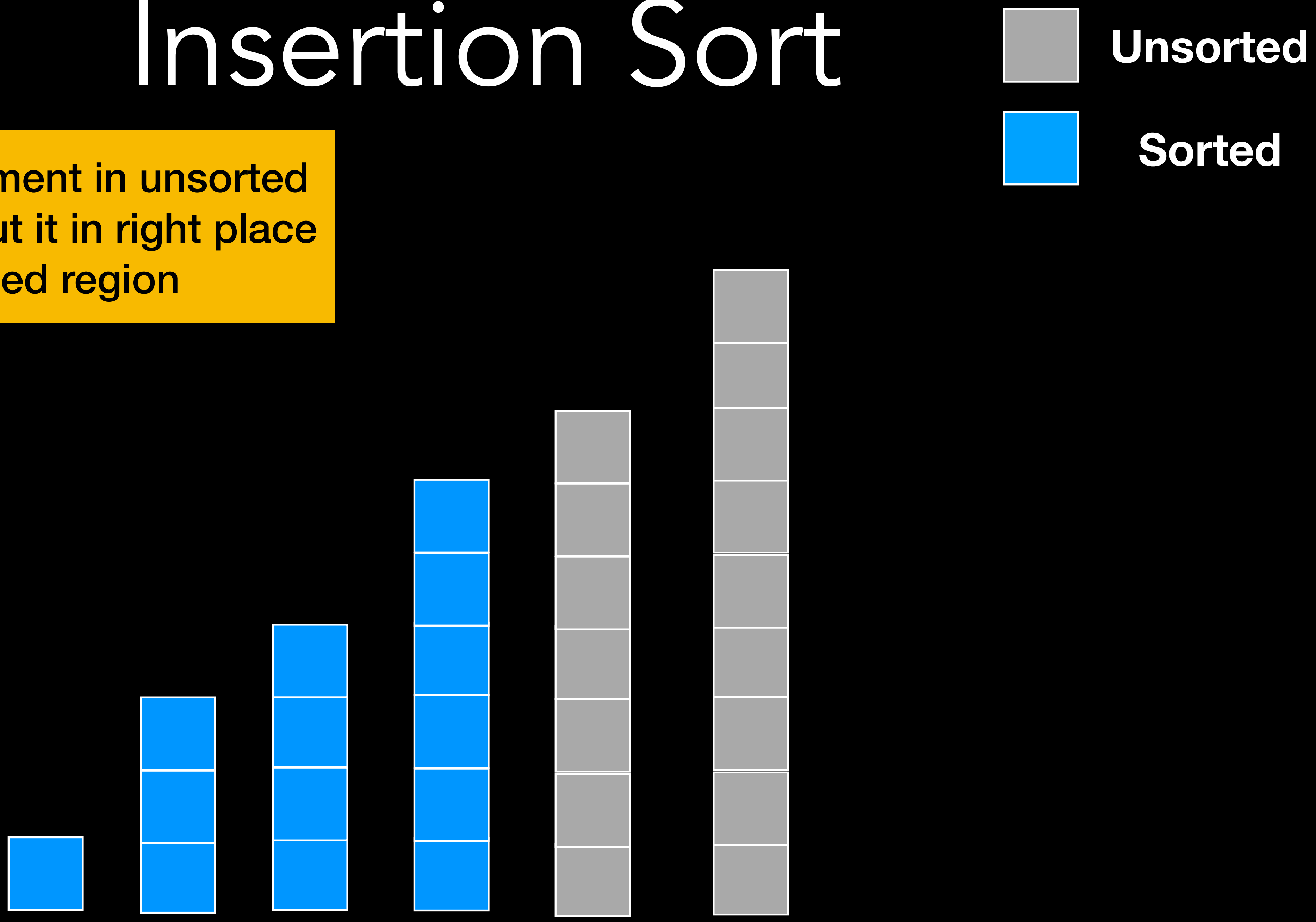
Sorted

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

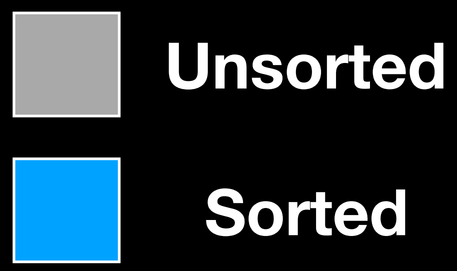


Insertion Sort

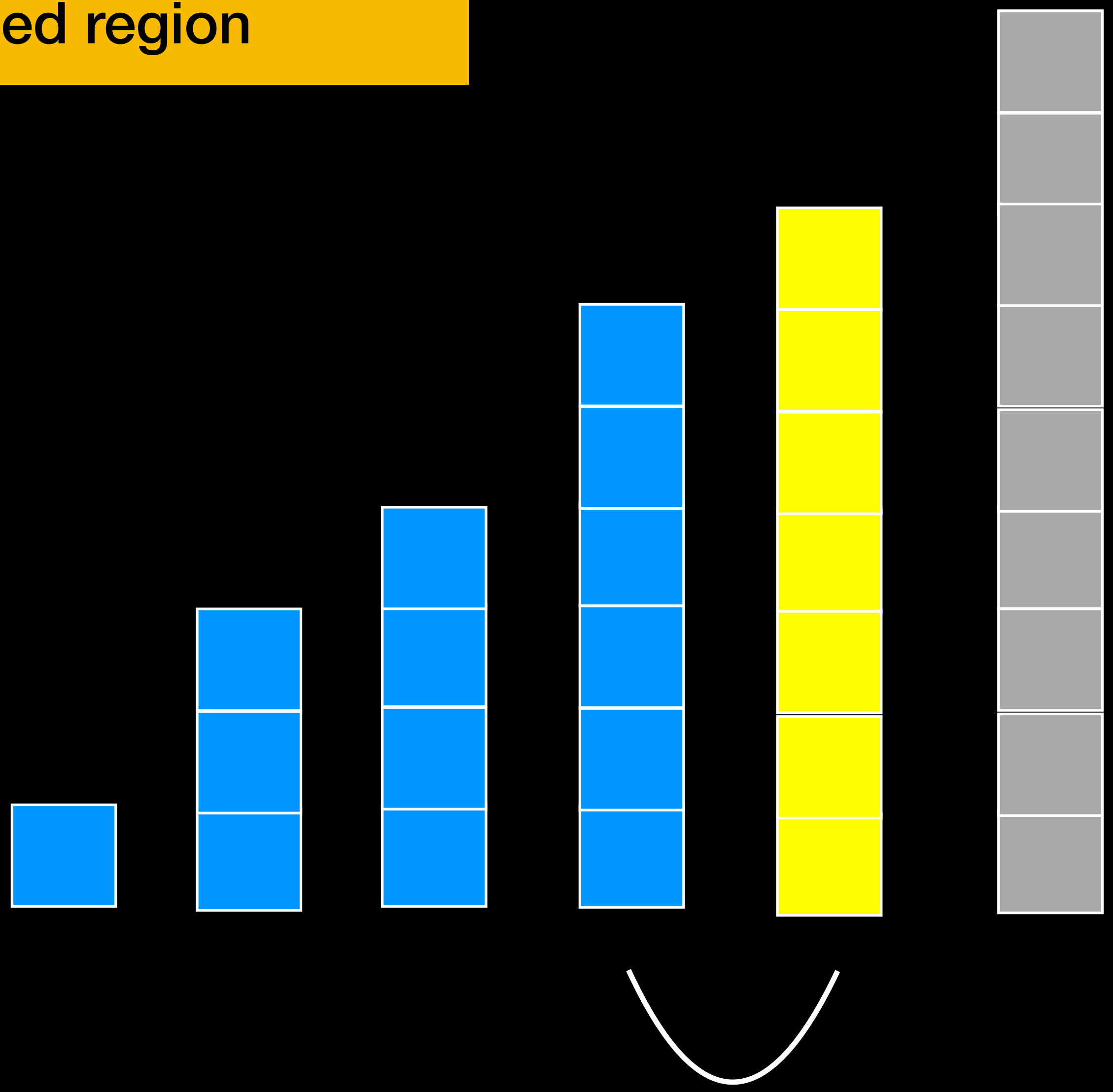
Pick first element in unsorted region and put it in right place in sorted region



Insertion Sort



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

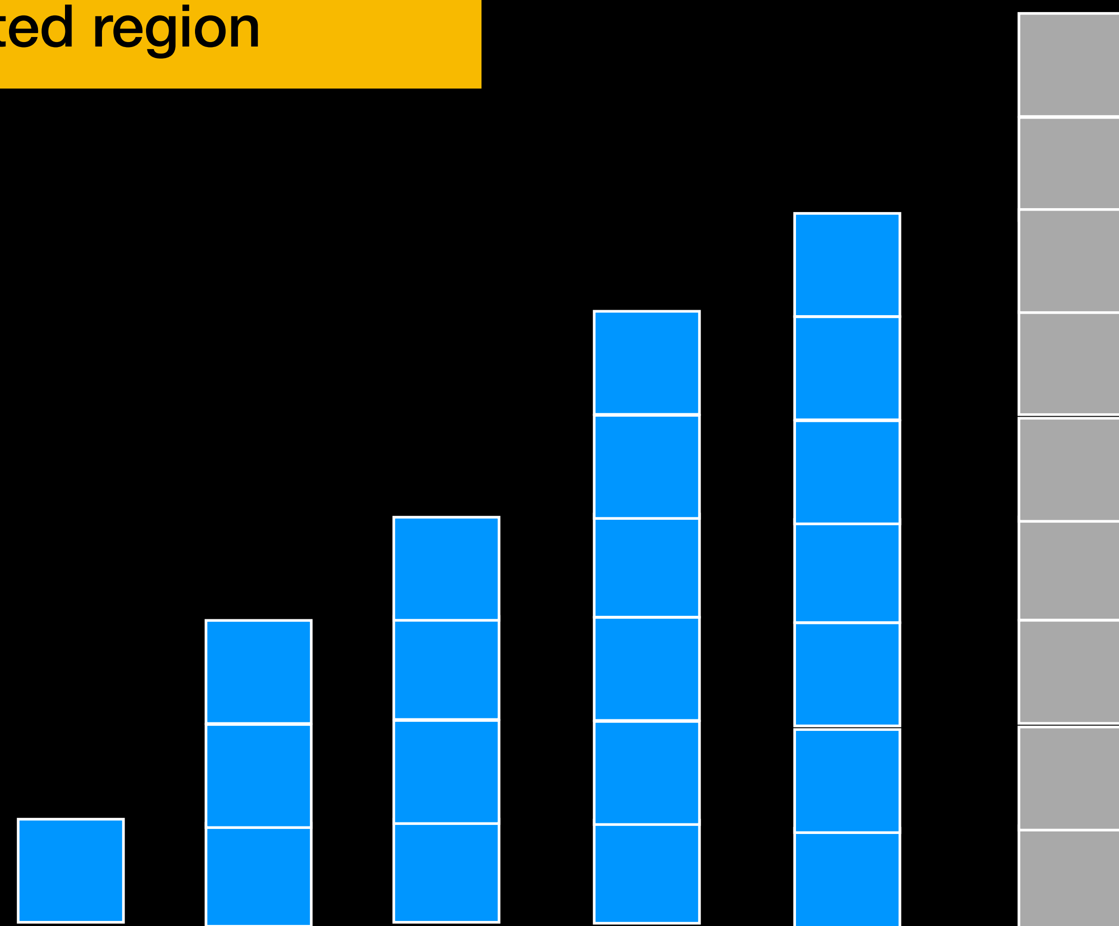


Insertion Sort

Unsorted

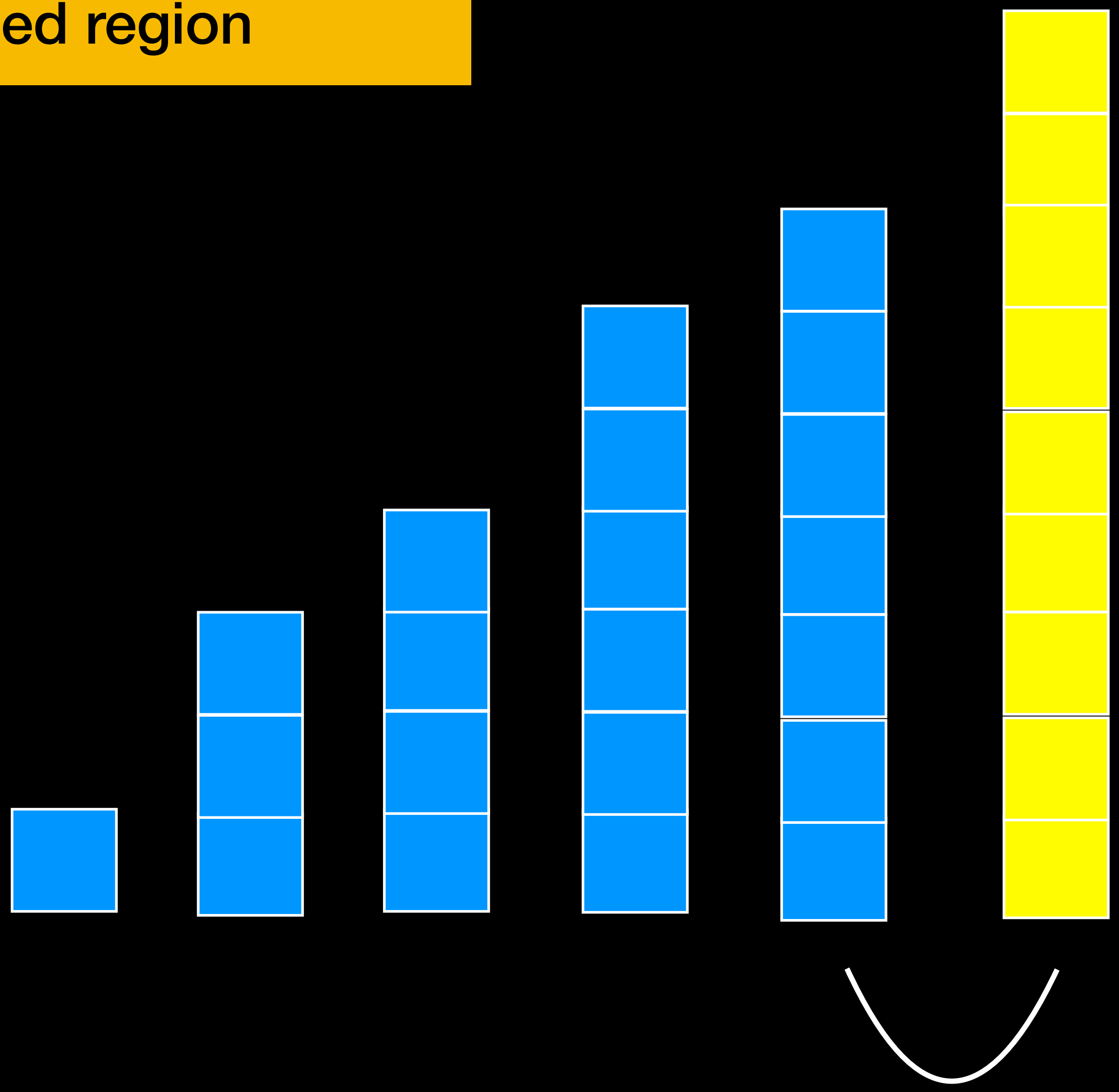
Sorted

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

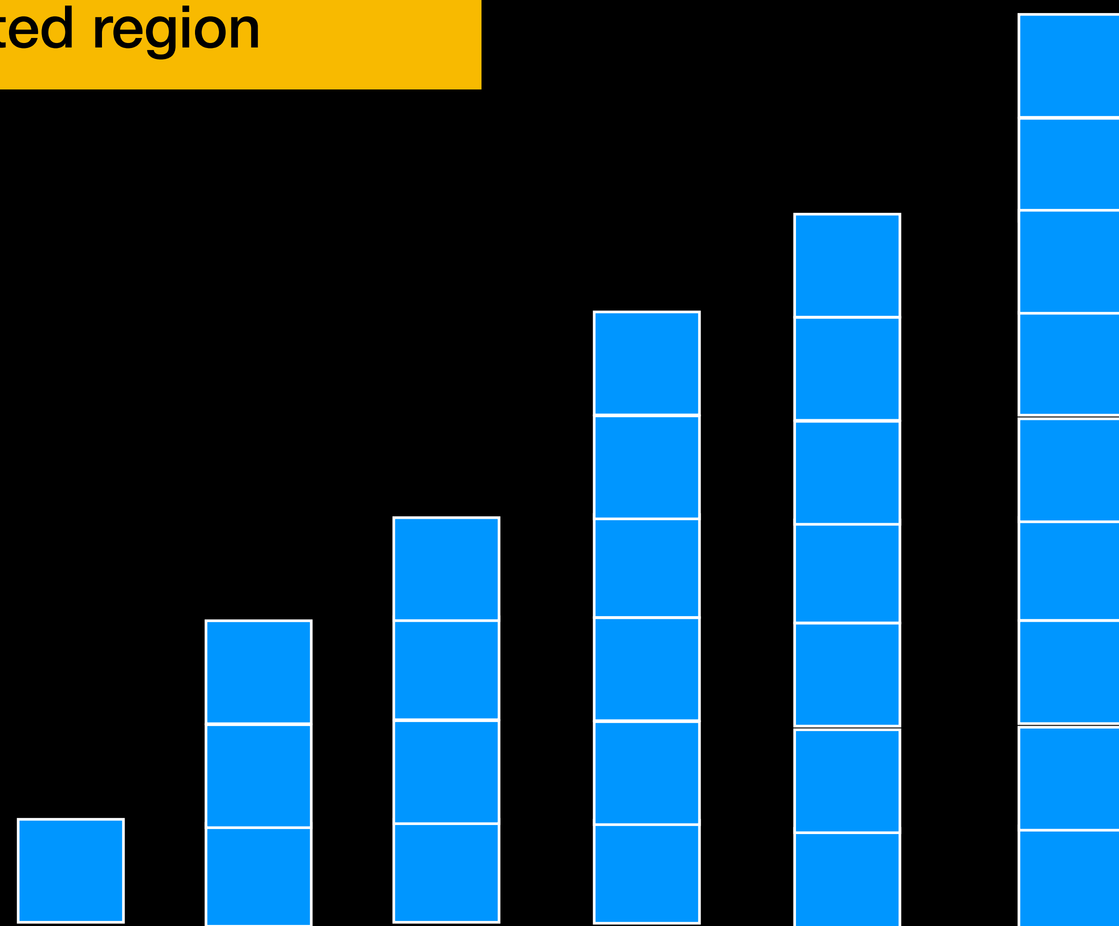


Insertion Sort

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

$O(n^2)$ comparisons and data moves

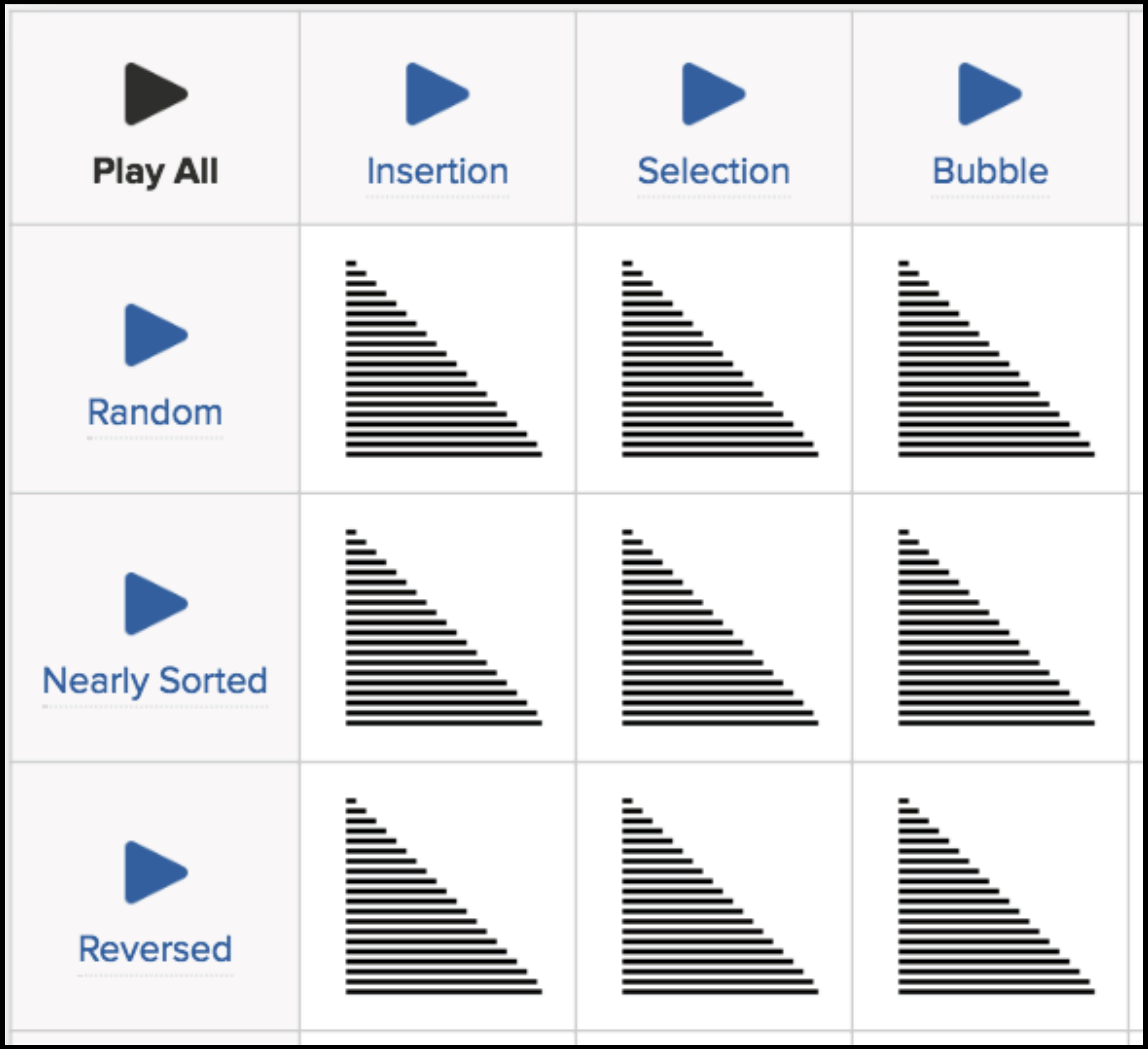
$\Omega(n)$

Stable

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

What we have so far

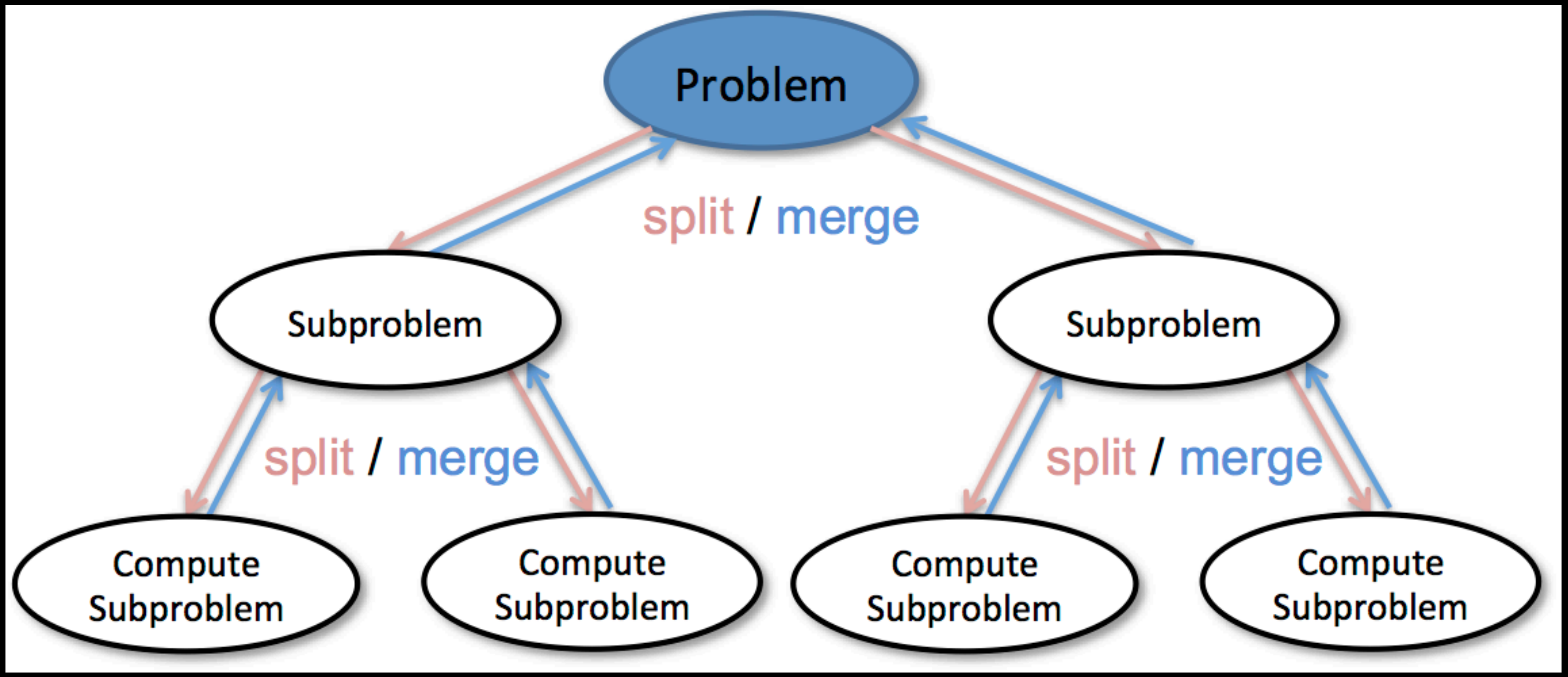
	O	Ω
Selection Sort	$O(n^2)$	$\Omega(n^2)$
Bubble Sort	$O(n^2)$	$\Omega(n)$
Insertion Sort	$O(n^2)$	$\Omega(n)$


Play All
Insertion
Selection
Bubble
Random
Nearly Sorted
Reversed

Can we do better?

Can we do better?

Divide and Conquer!!!



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

195	599	158	2	260	11	64	932	5

$T(^1/_2 n)$ $T(^1/_2 n)$

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

$T(^1/_2 n)$

195	599	158	2	260	11	64	932	5

$T(^1/_2 n)$

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

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

$T(^1/_2 n) \approx {}^1/_4\, T(n)$ $T(^1/_2 n) \approx {}^1/_4\, T(n)$

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

Understanding $O(n^2)$

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

$T(n)$

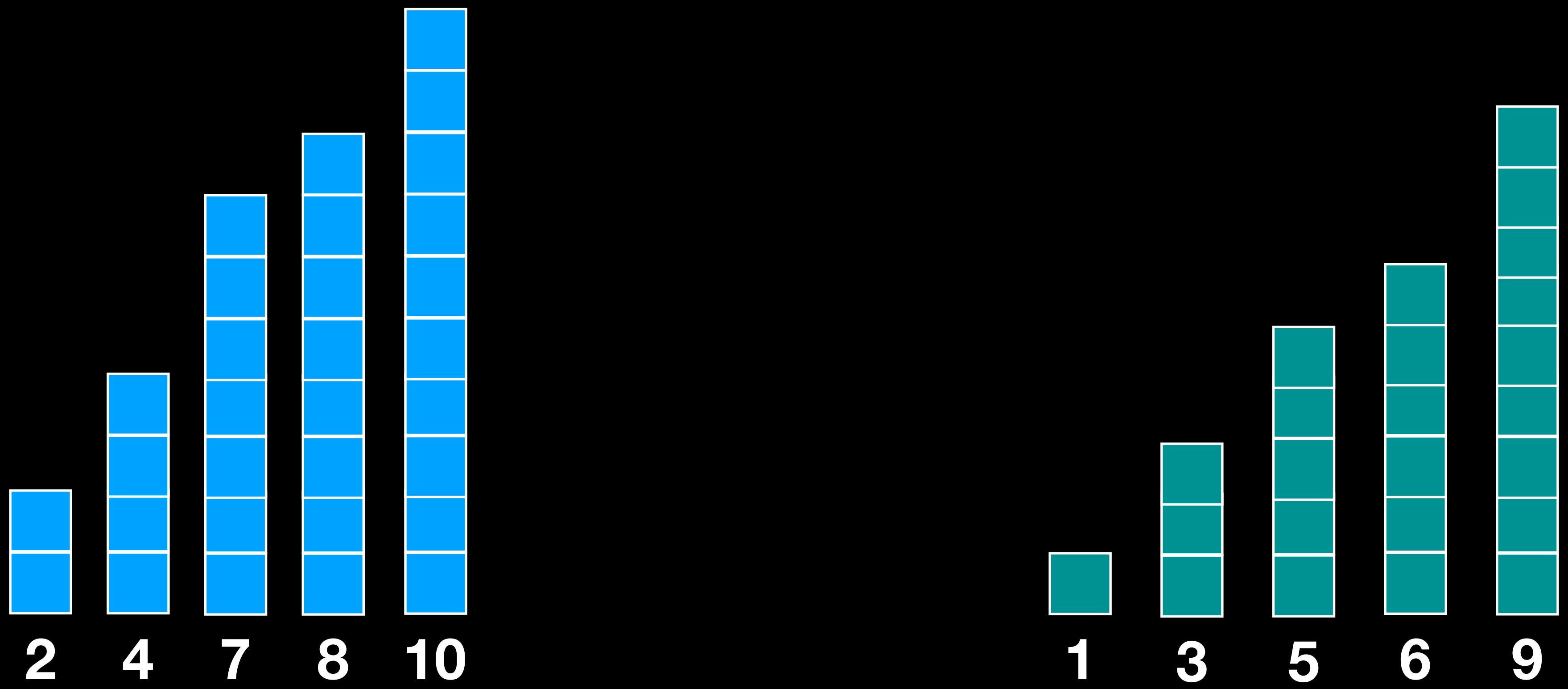
3	14	43	76	100	108	200	274	523

2	5	11	64	158	195	260	599	932

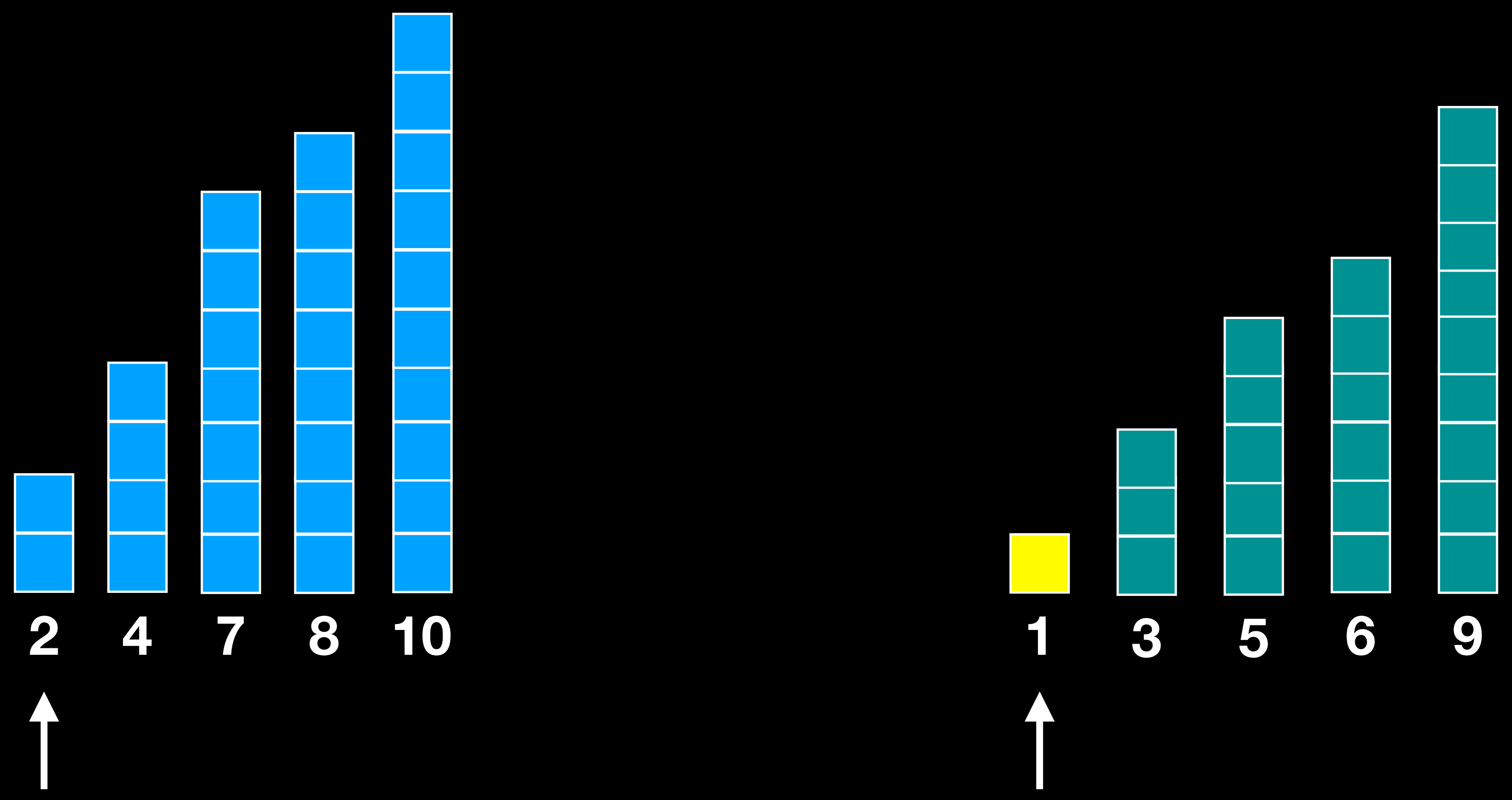
$T(^1/_2 n) \approx {}^1/_4\, T(n)$ $T(^1/_2 n) \approx {}^1/_4\, T(n)$

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

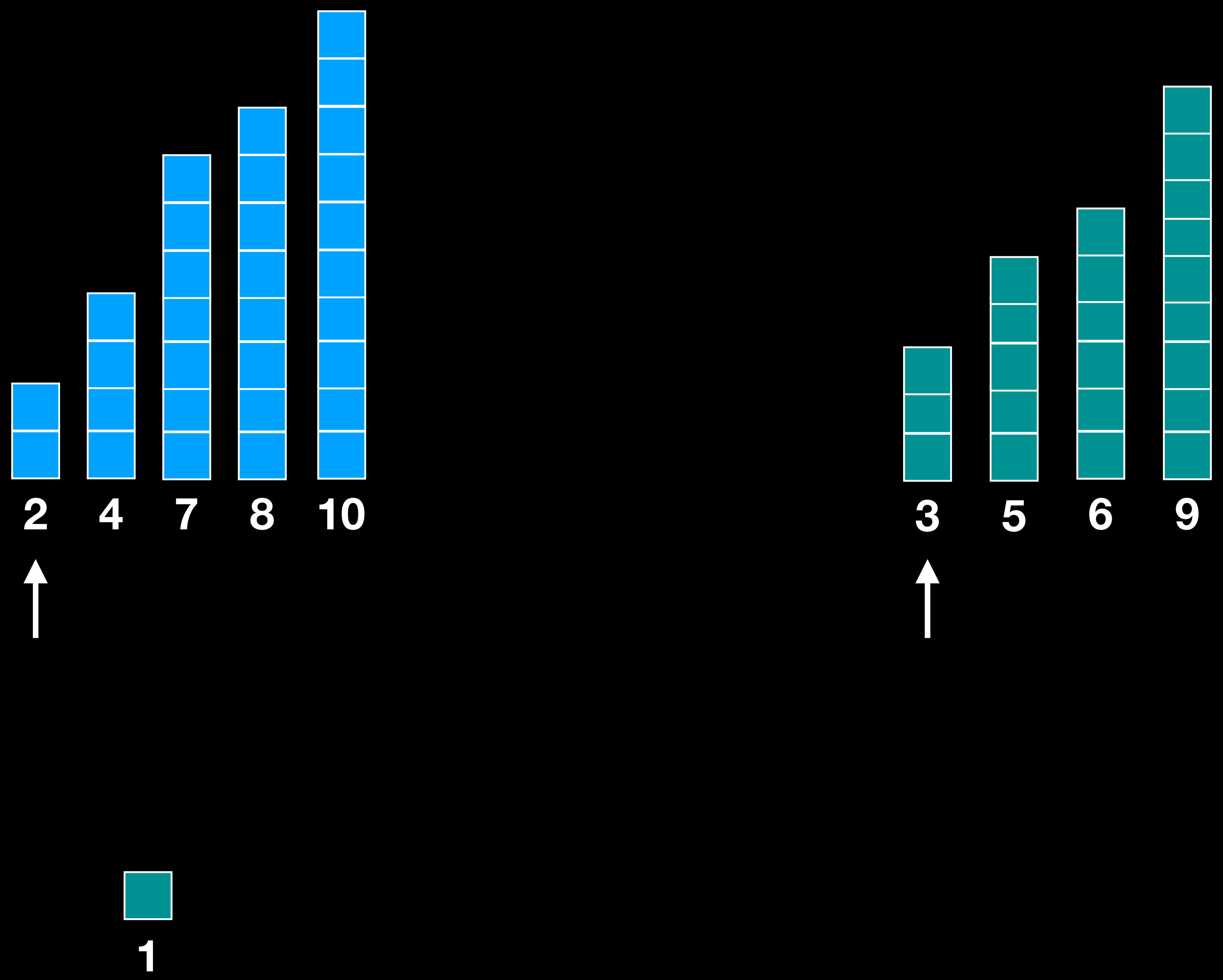
Key Insight: Merge is linear



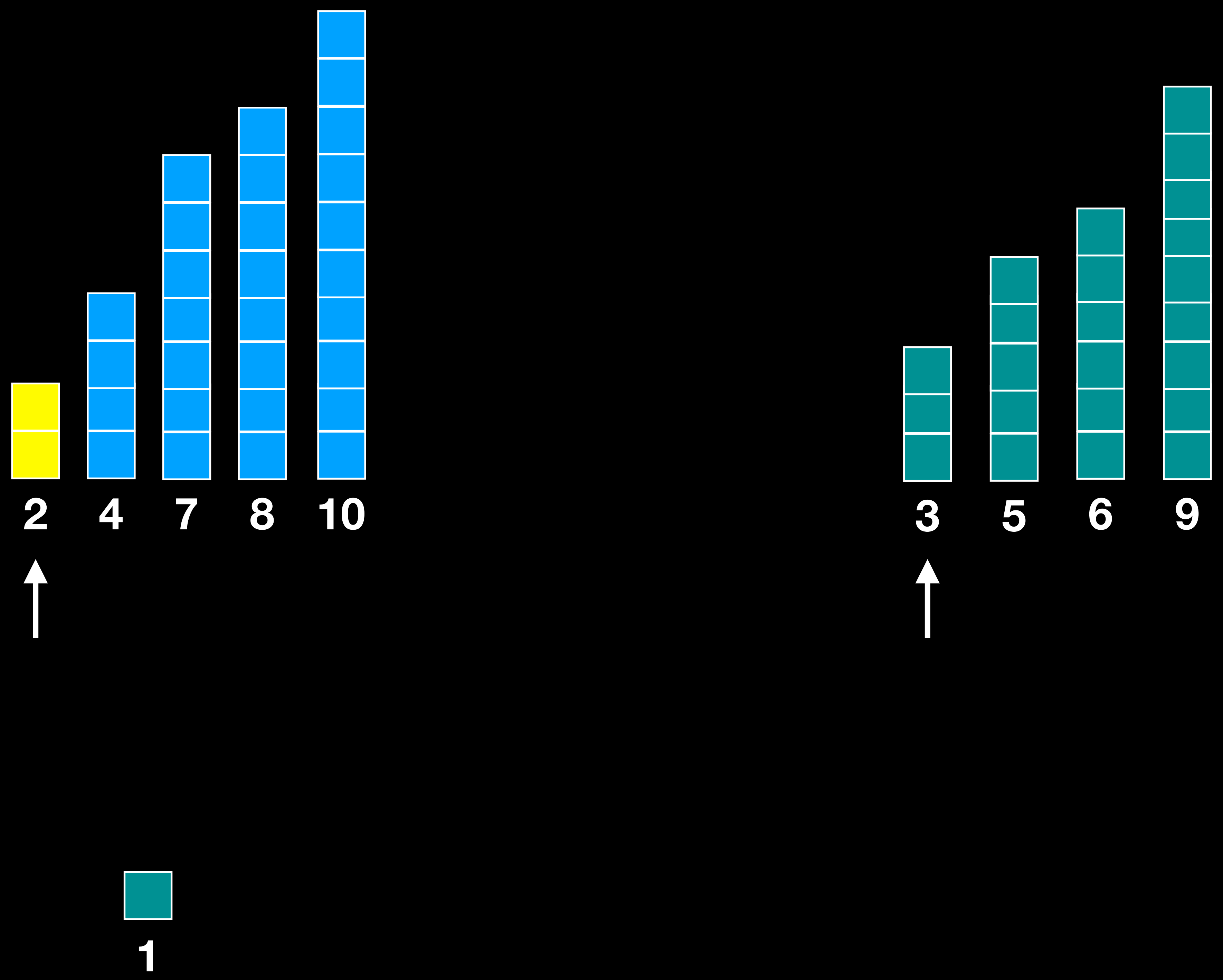
Key Insight: Merge is linear



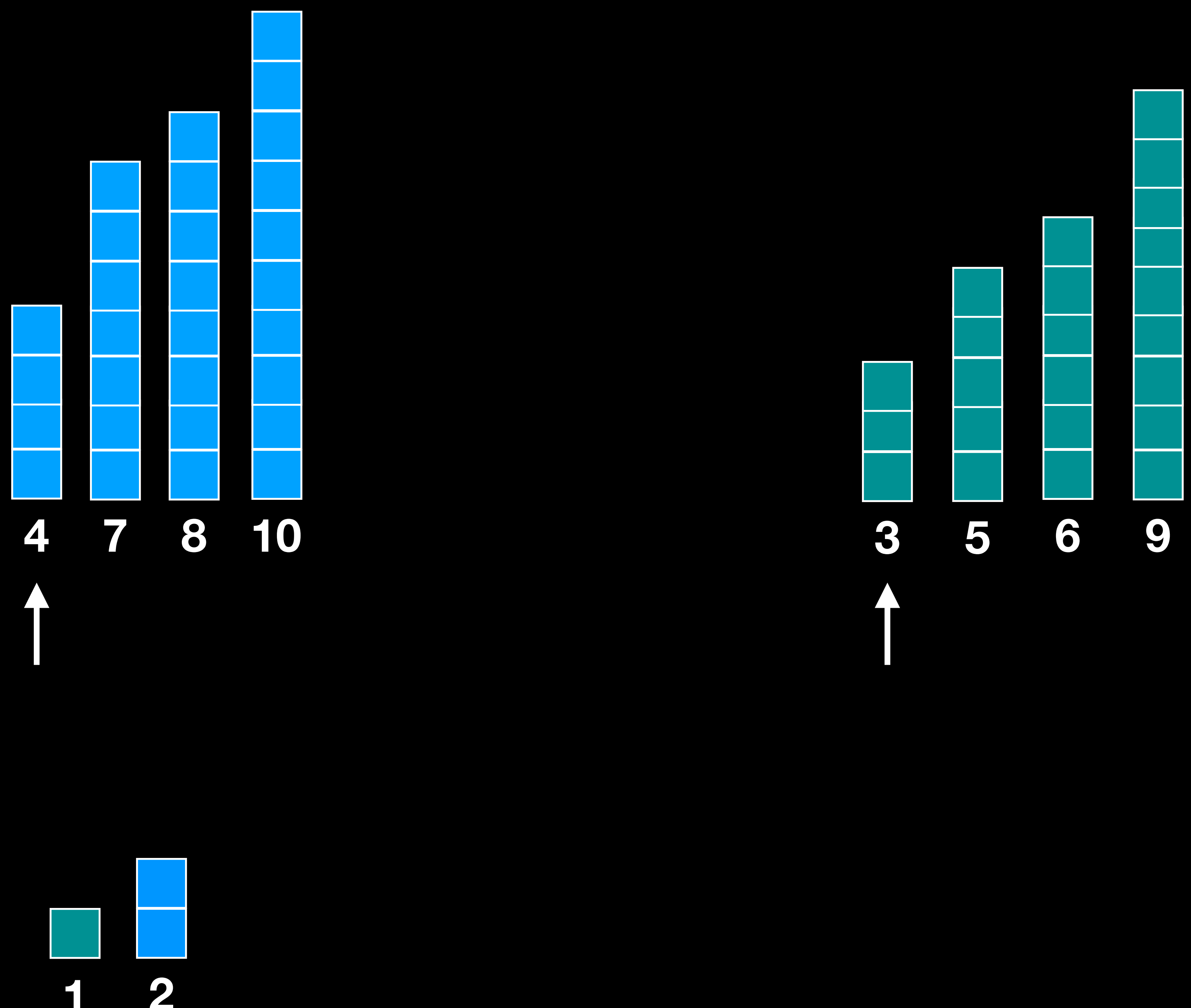
Key Insight: Merge is linear



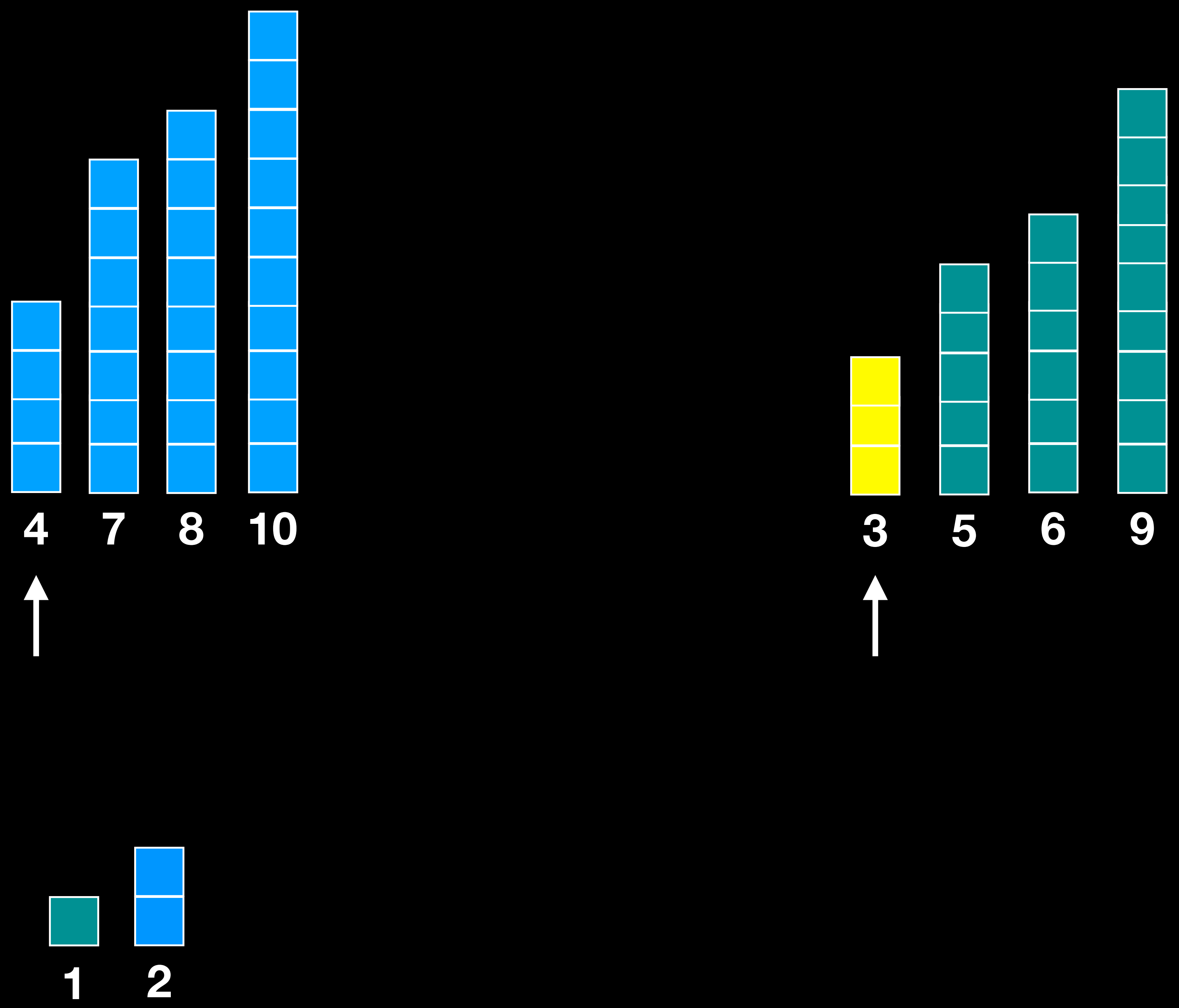
Key Insight: Merge is linear



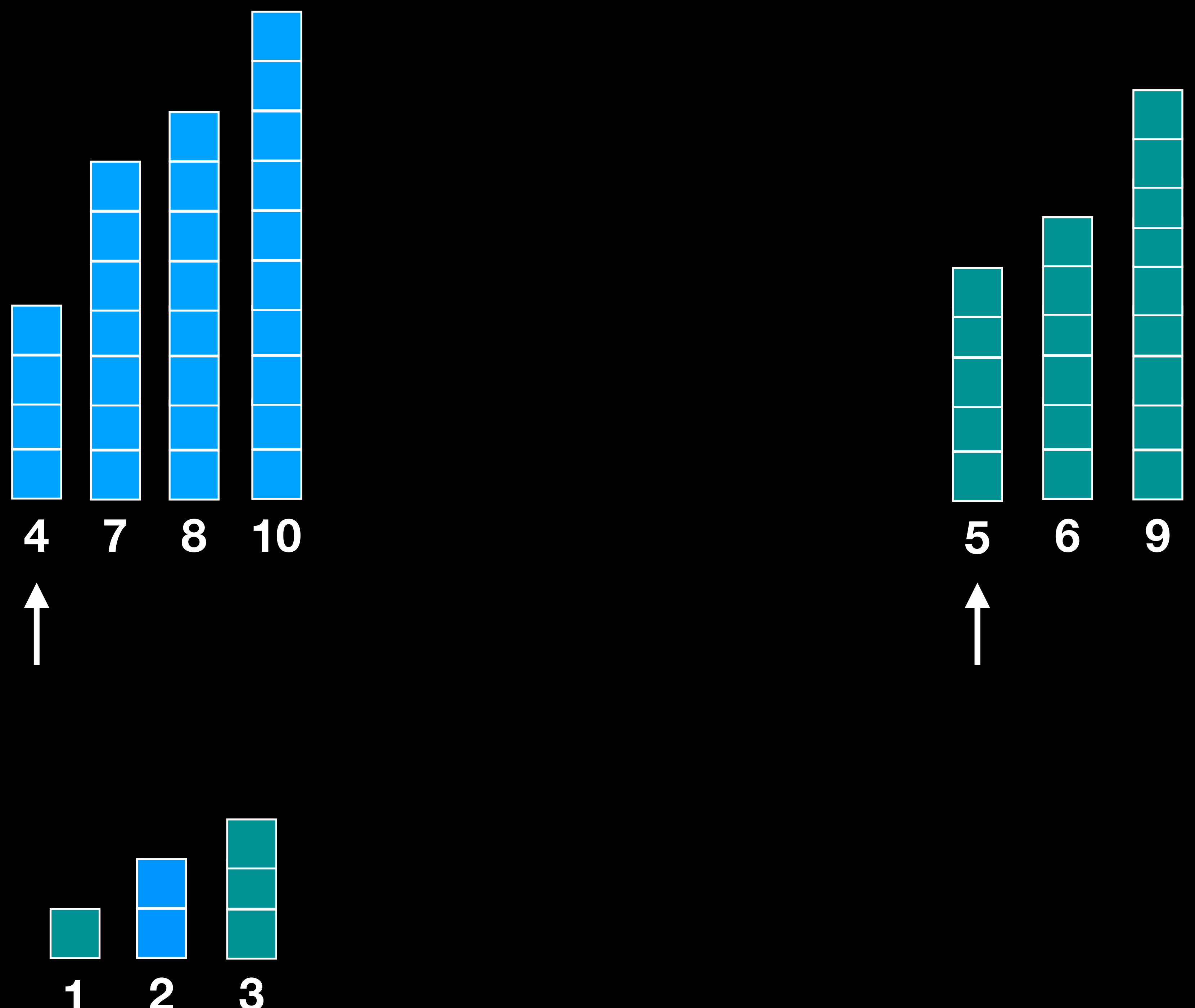
Key Insight: Merge is linear



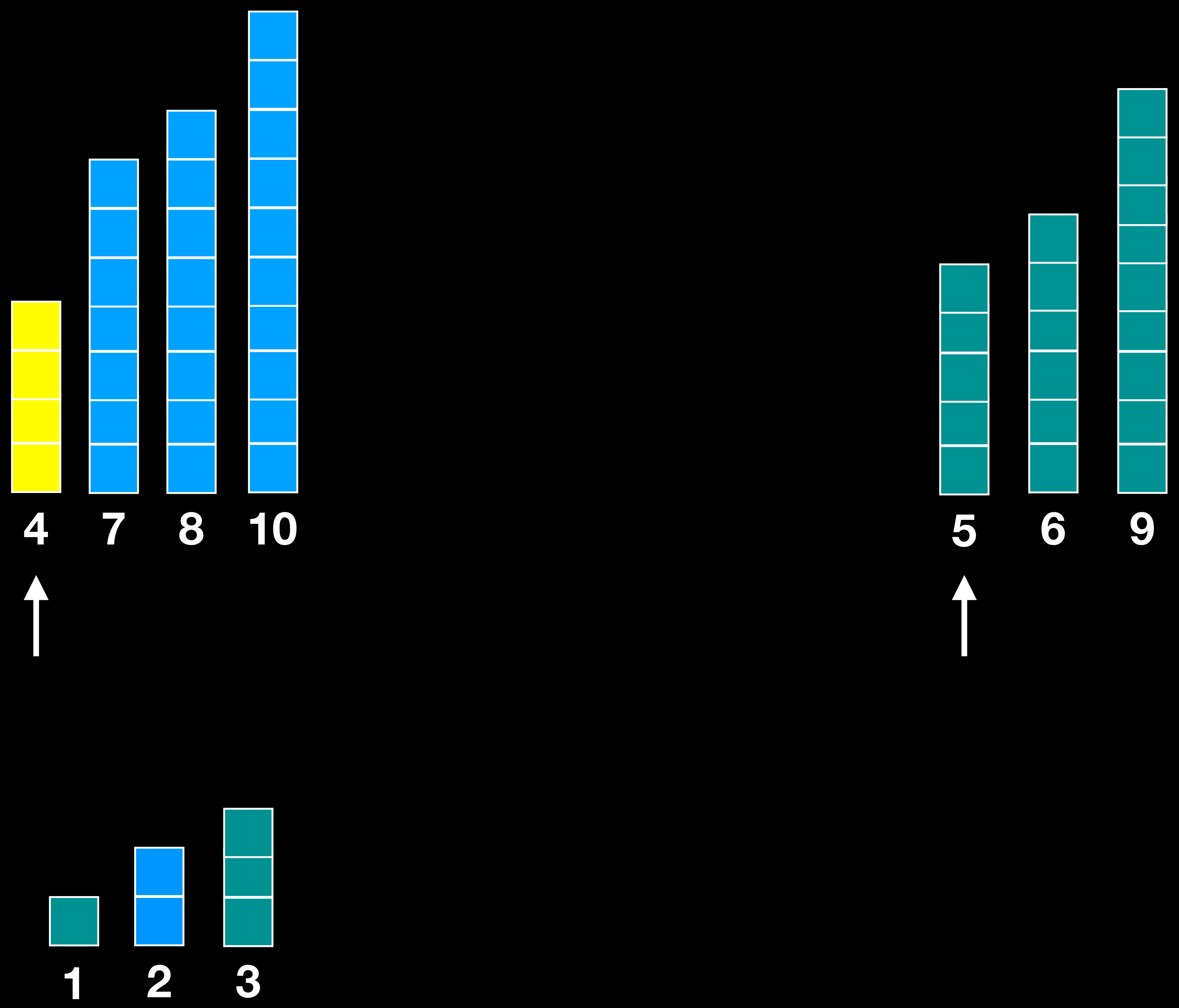
Key Insight: Merge is linear



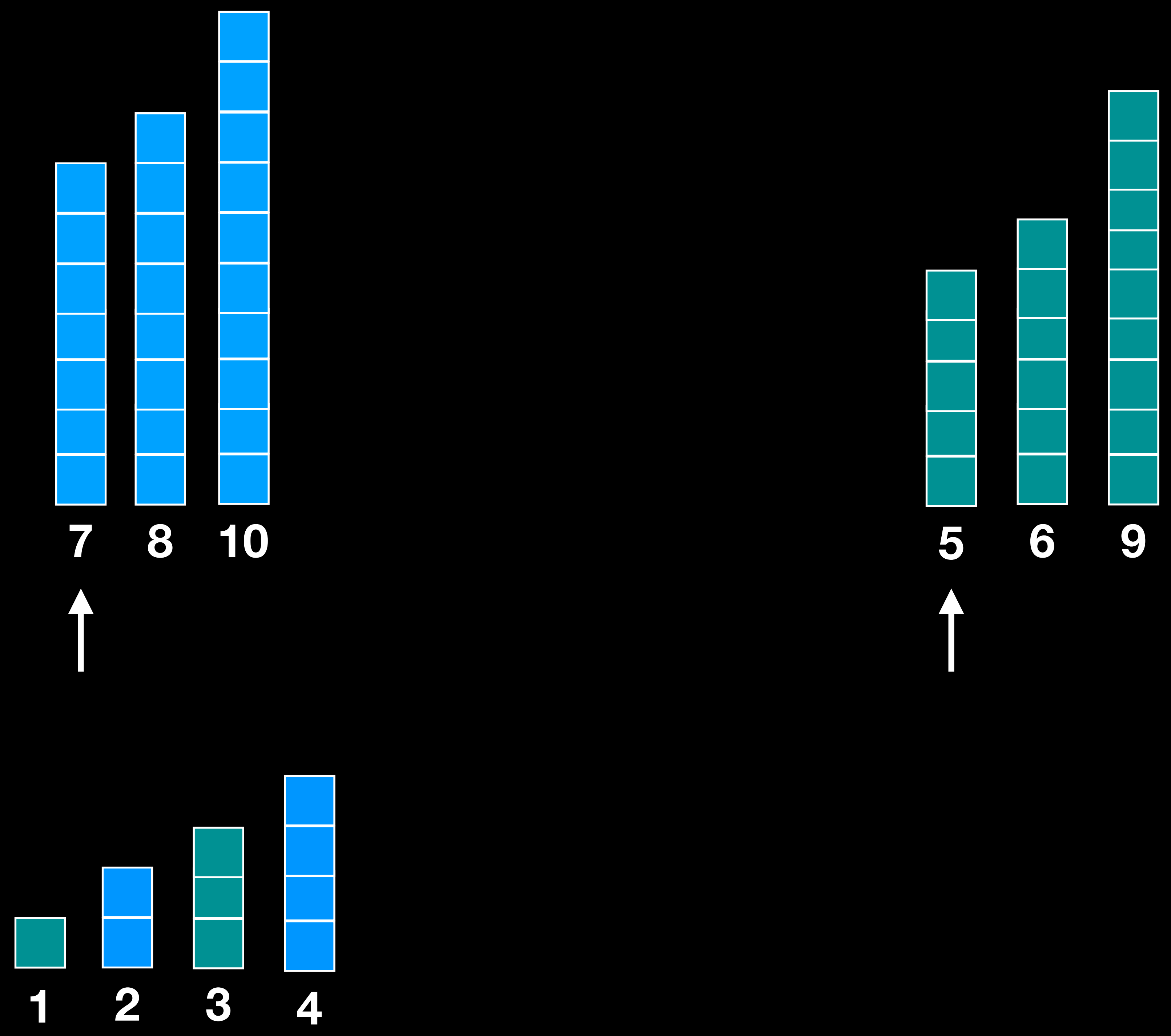
Key Insight: Merge is linear



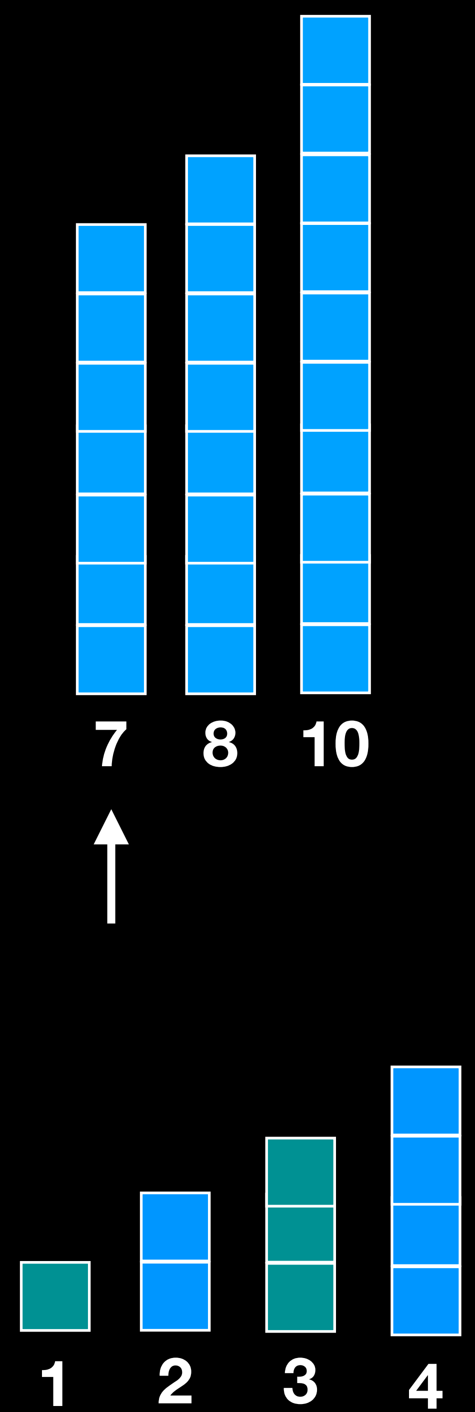
Key Insight: Merge is linear



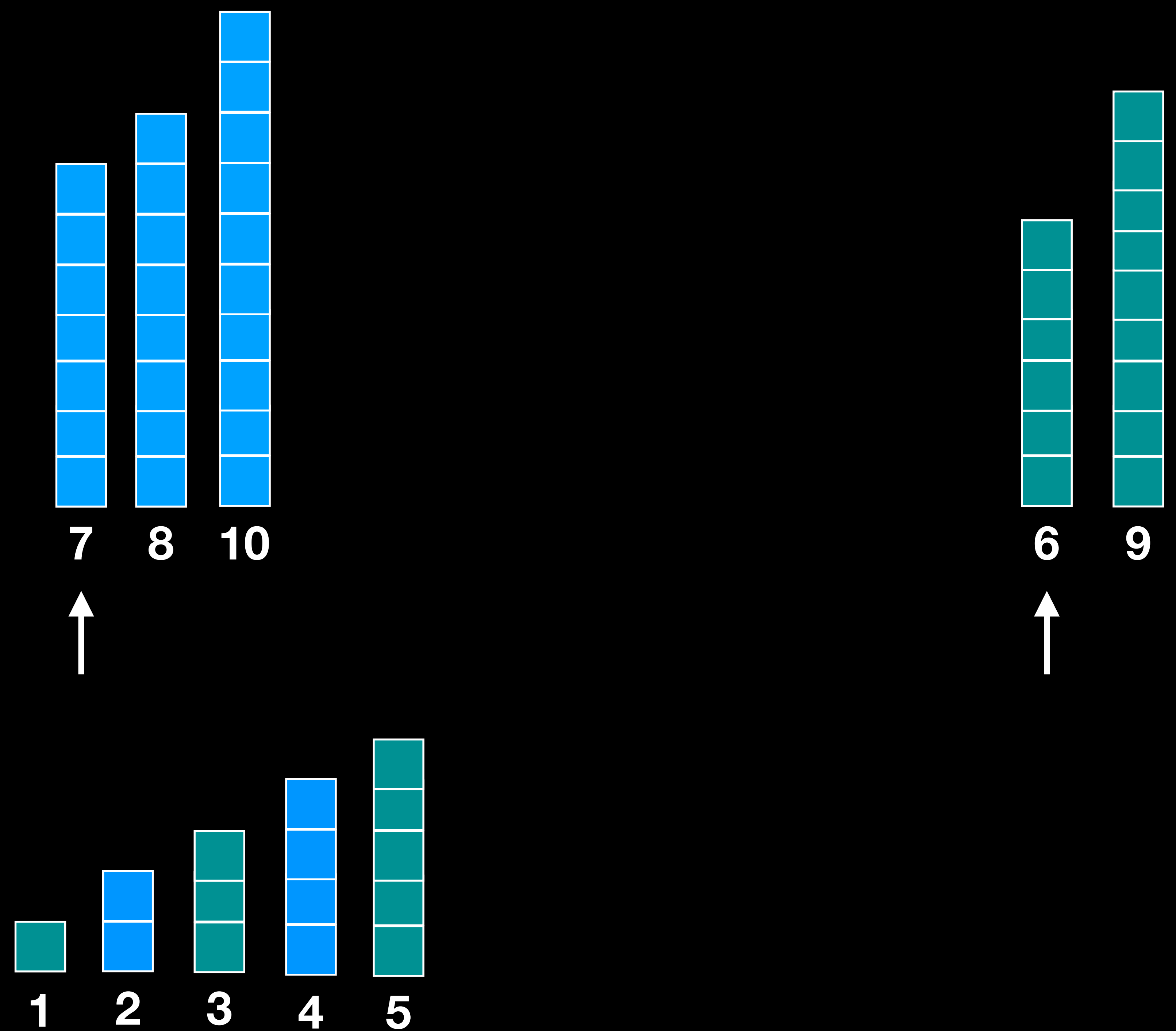
Key Insight: Merge is linear



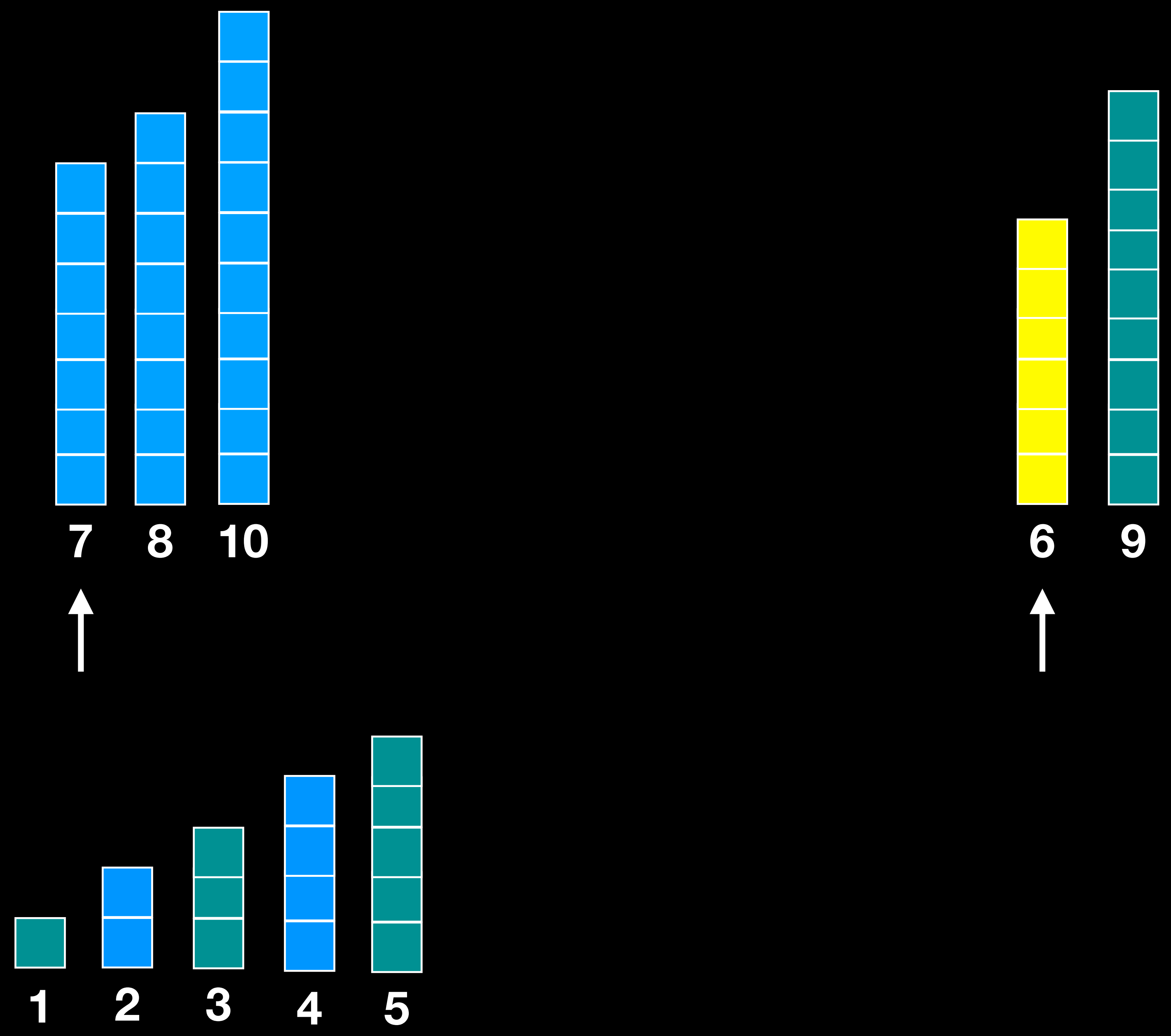
Key Insight: Merge is linear



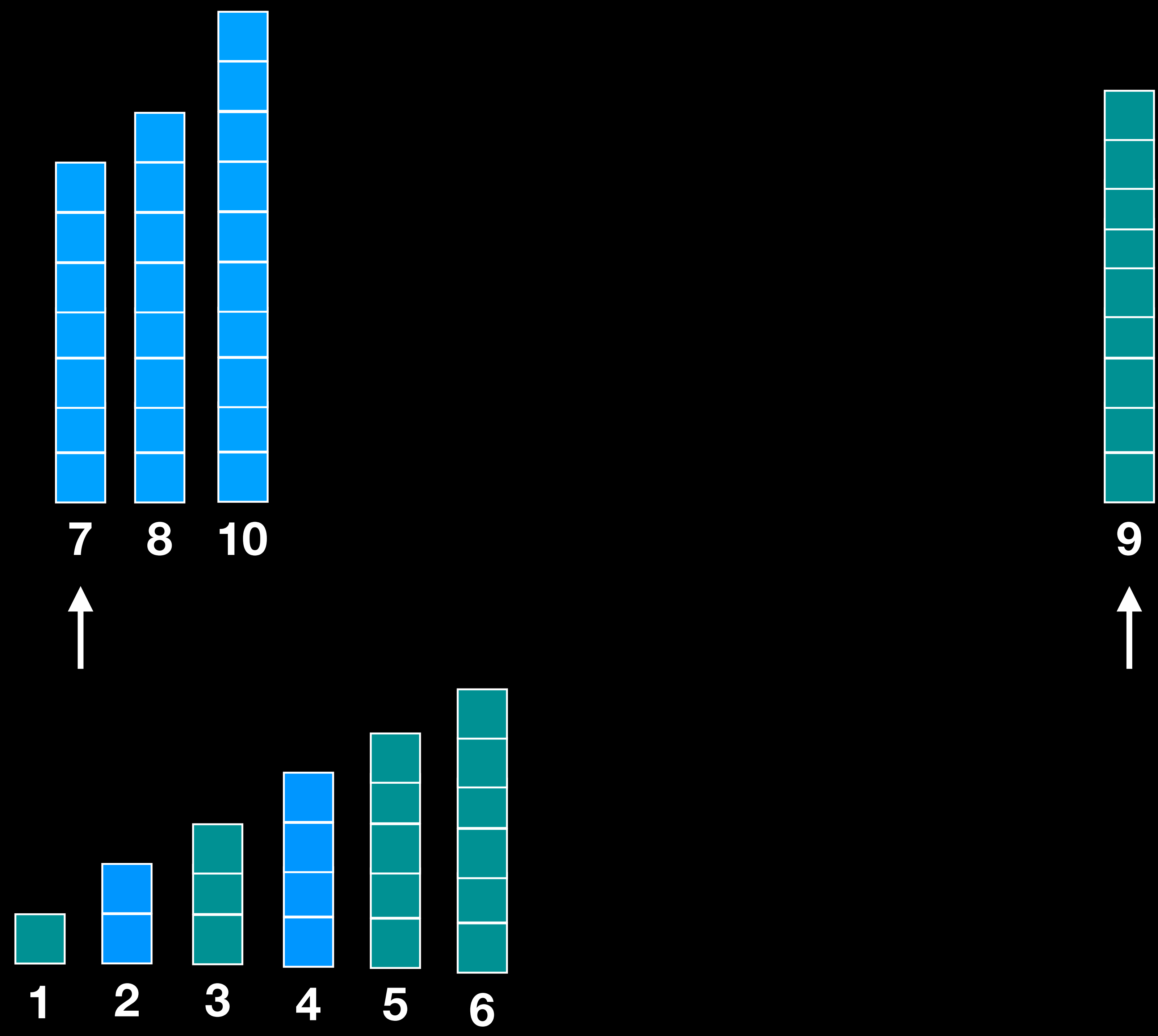
Key Insight: Merge is linear



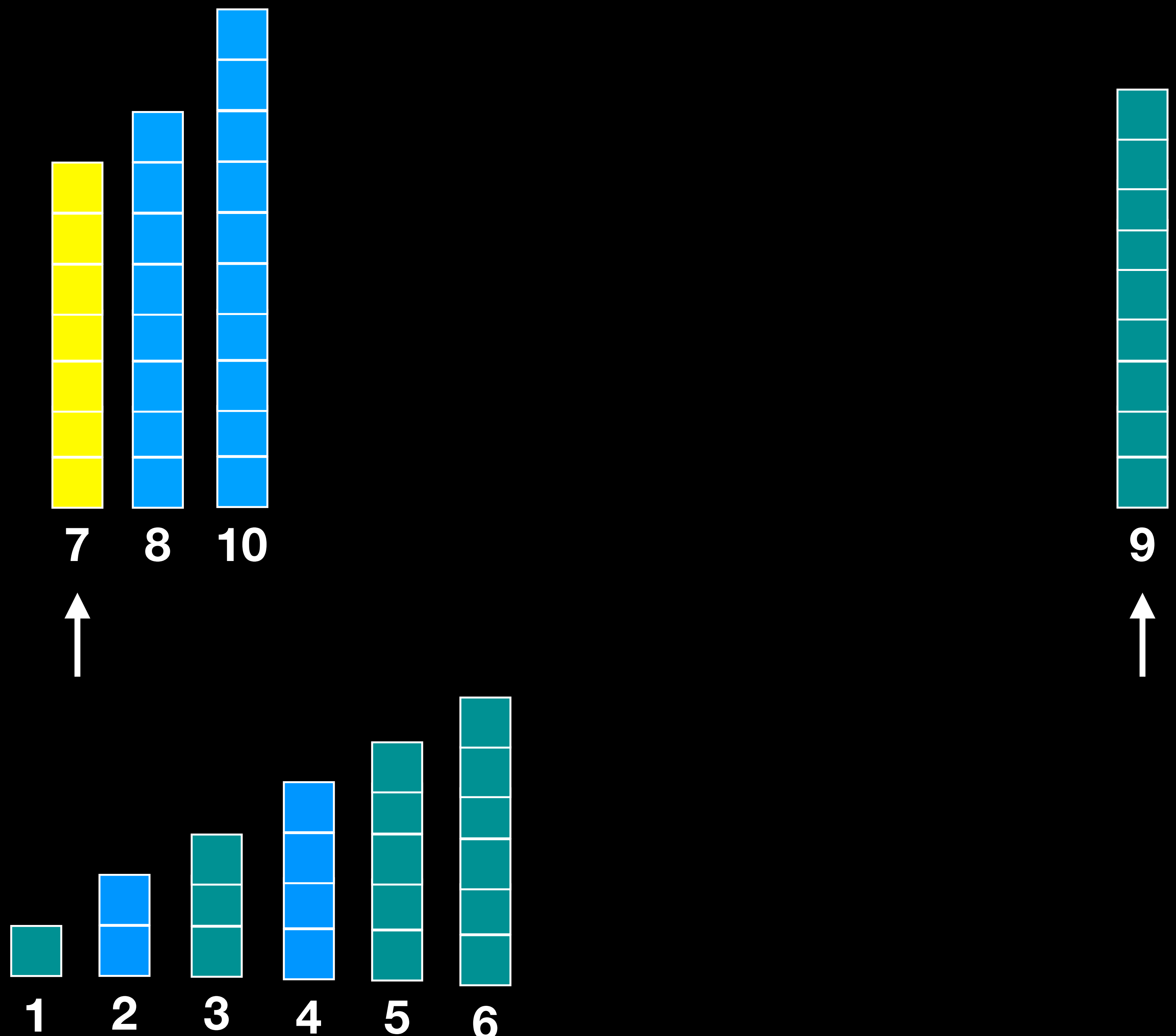
Key Insight: Merge is linear



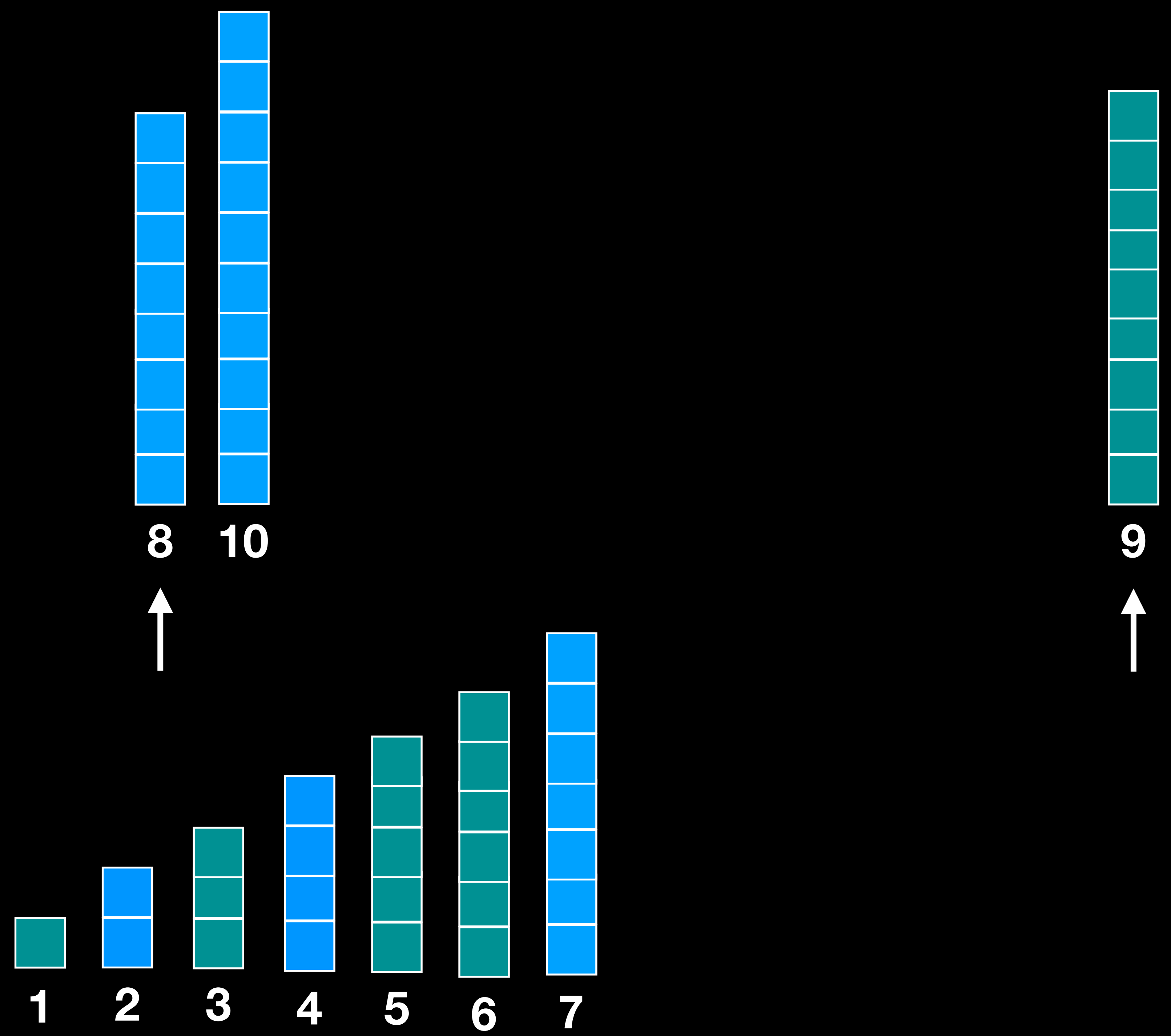
Key Insight: Merge is linear



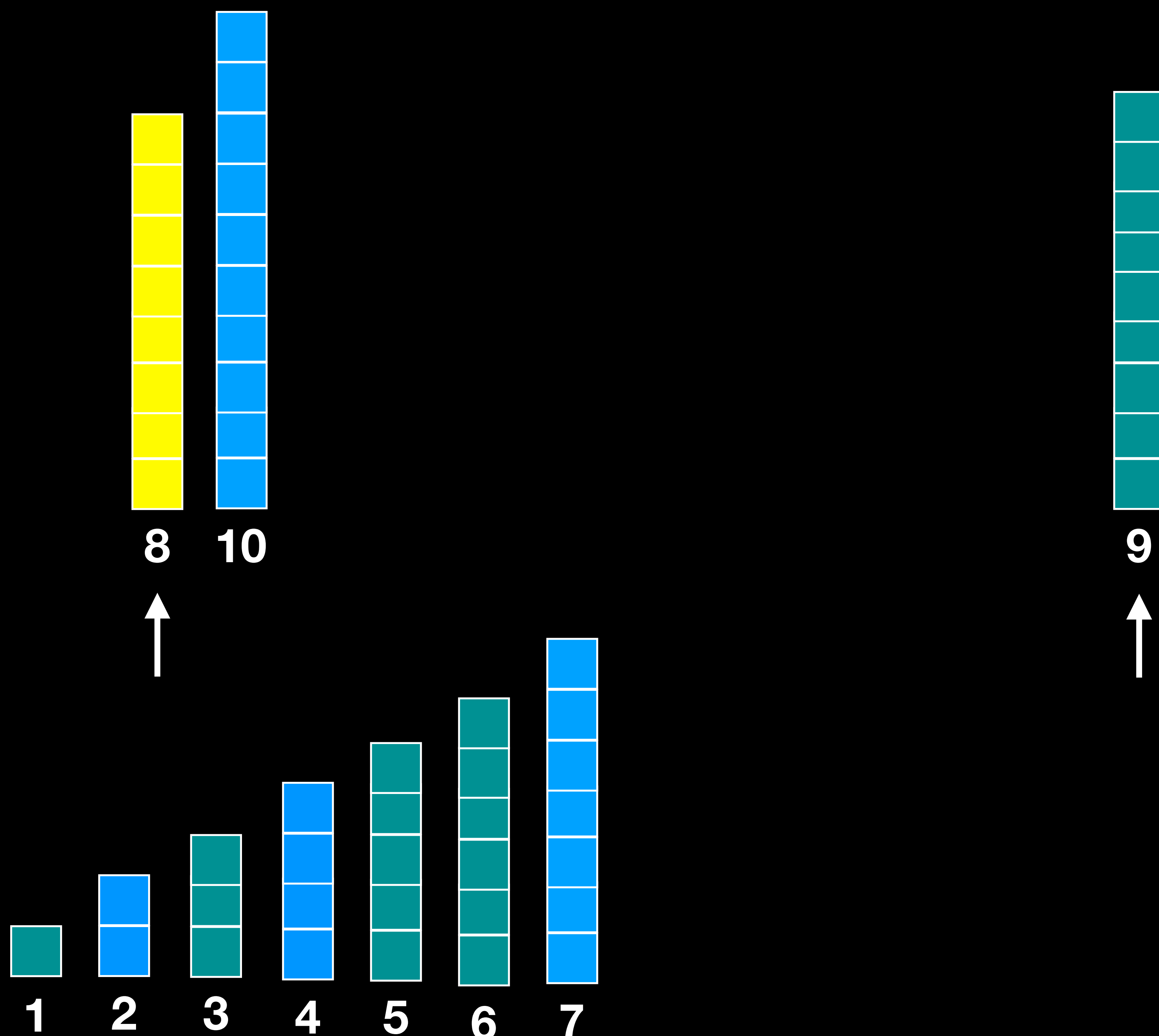
Key Insight: Merge is linear



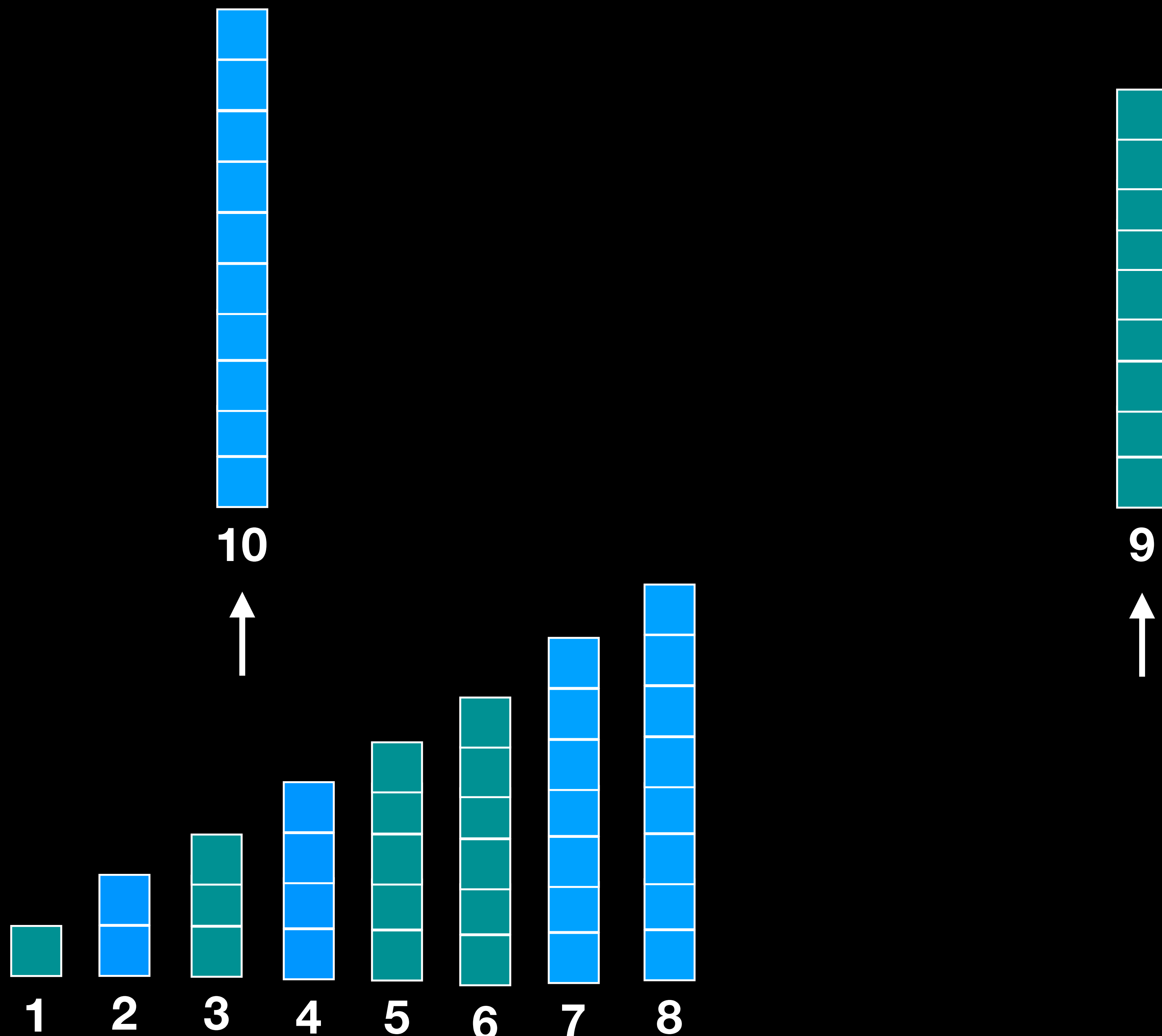
Key Insight: Merge is linear



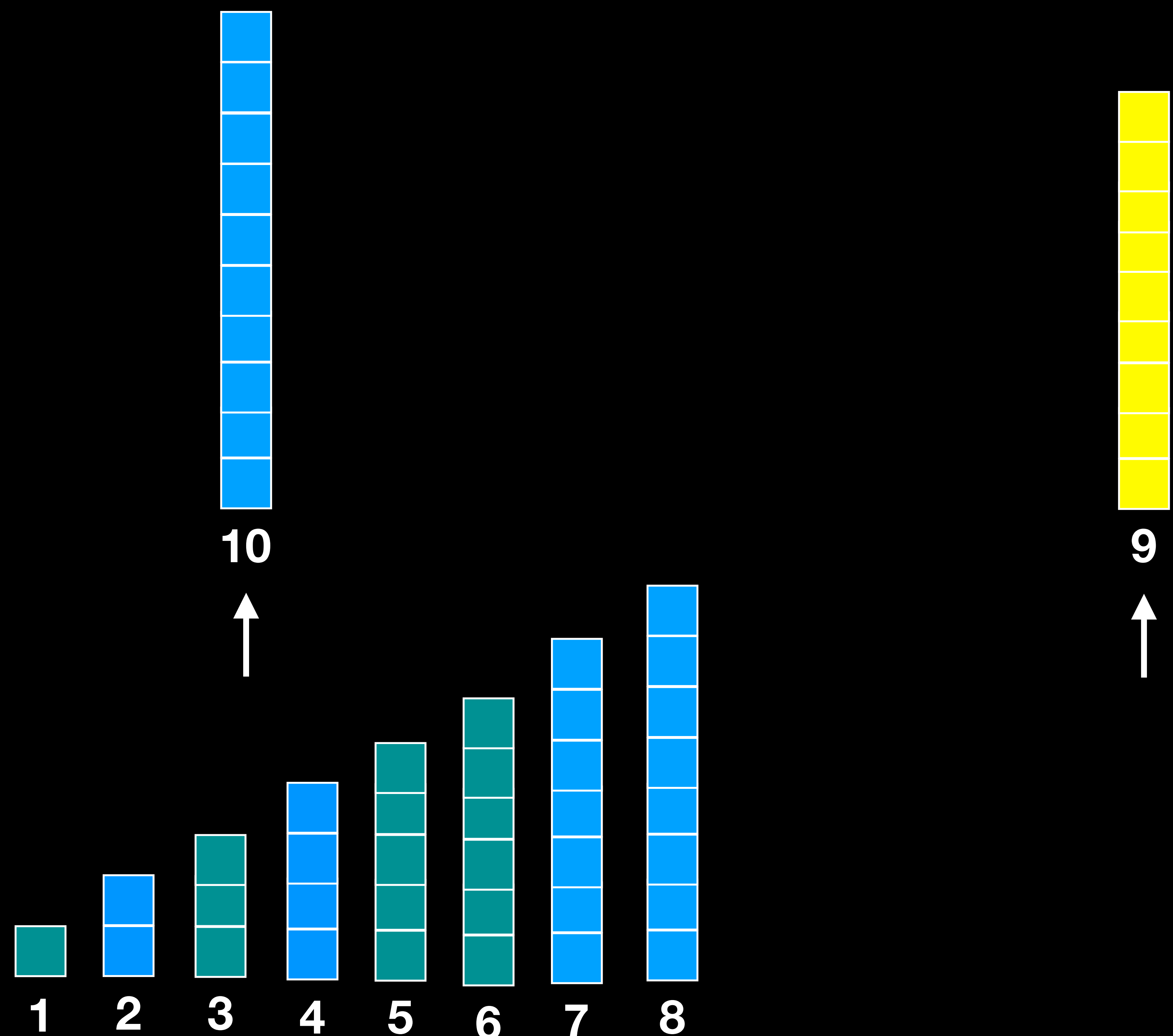
Key Insight: Merge is linear



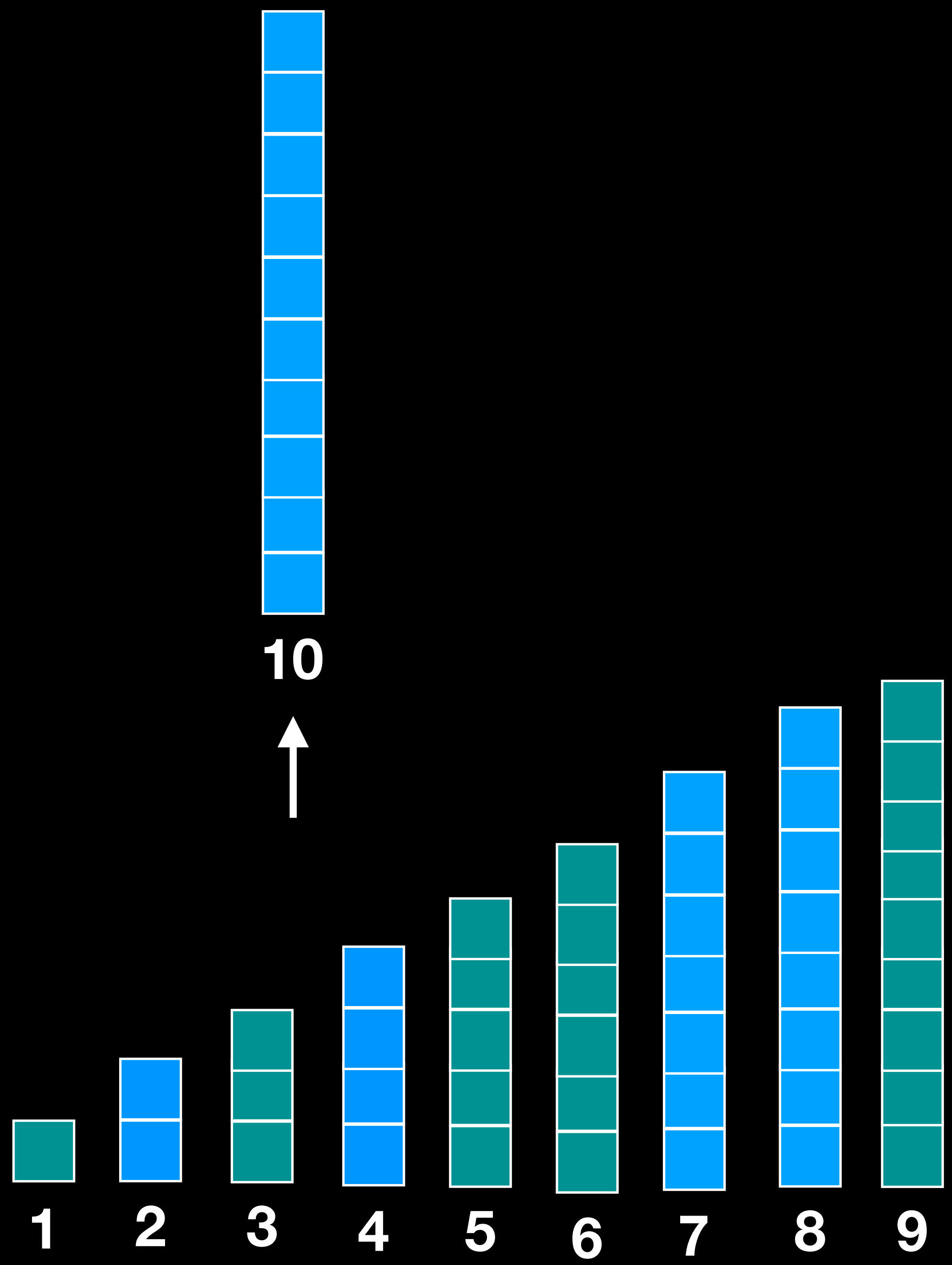
Key Insight: Merge is linear



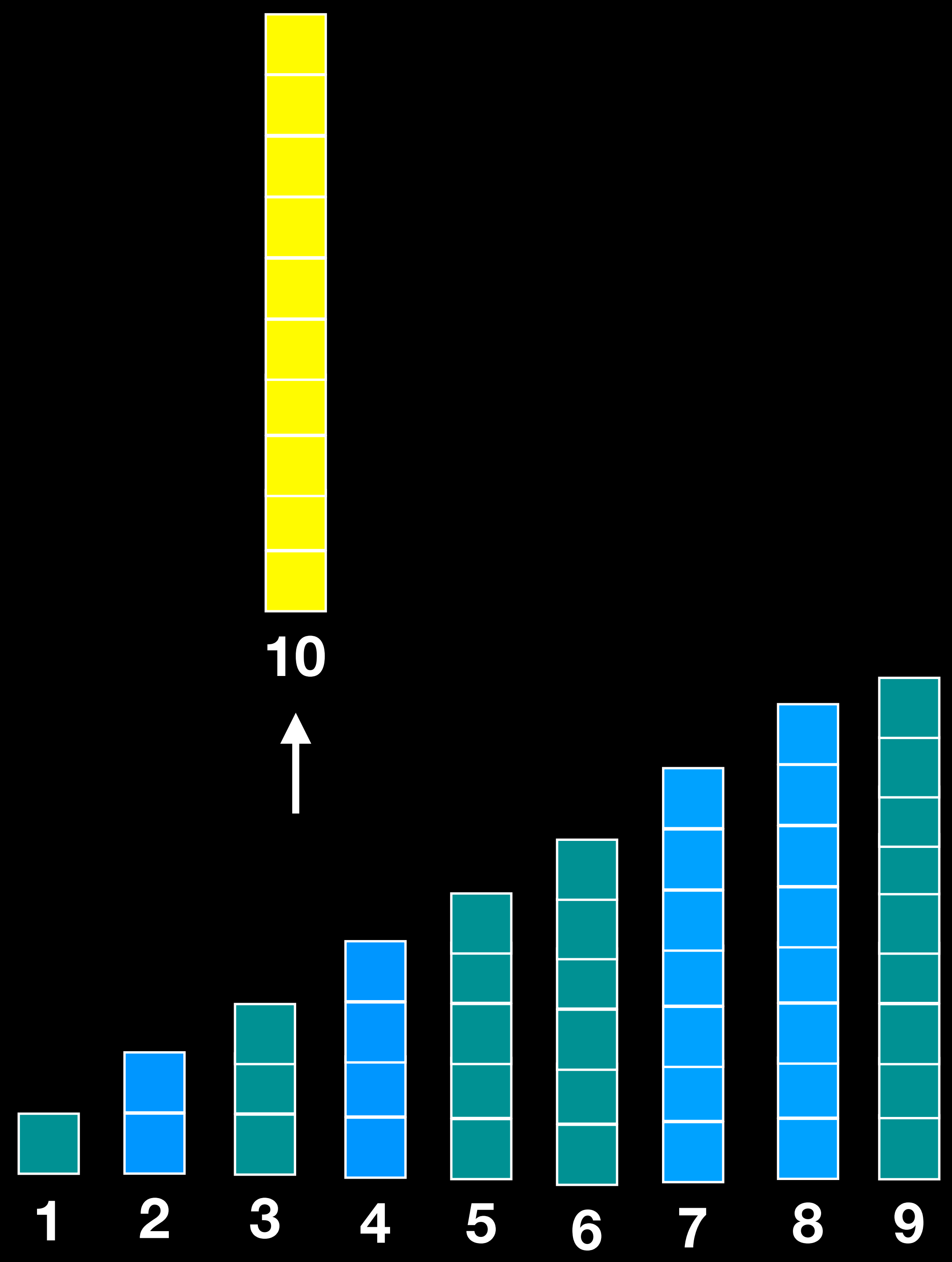
Key Insight: Merge is linear



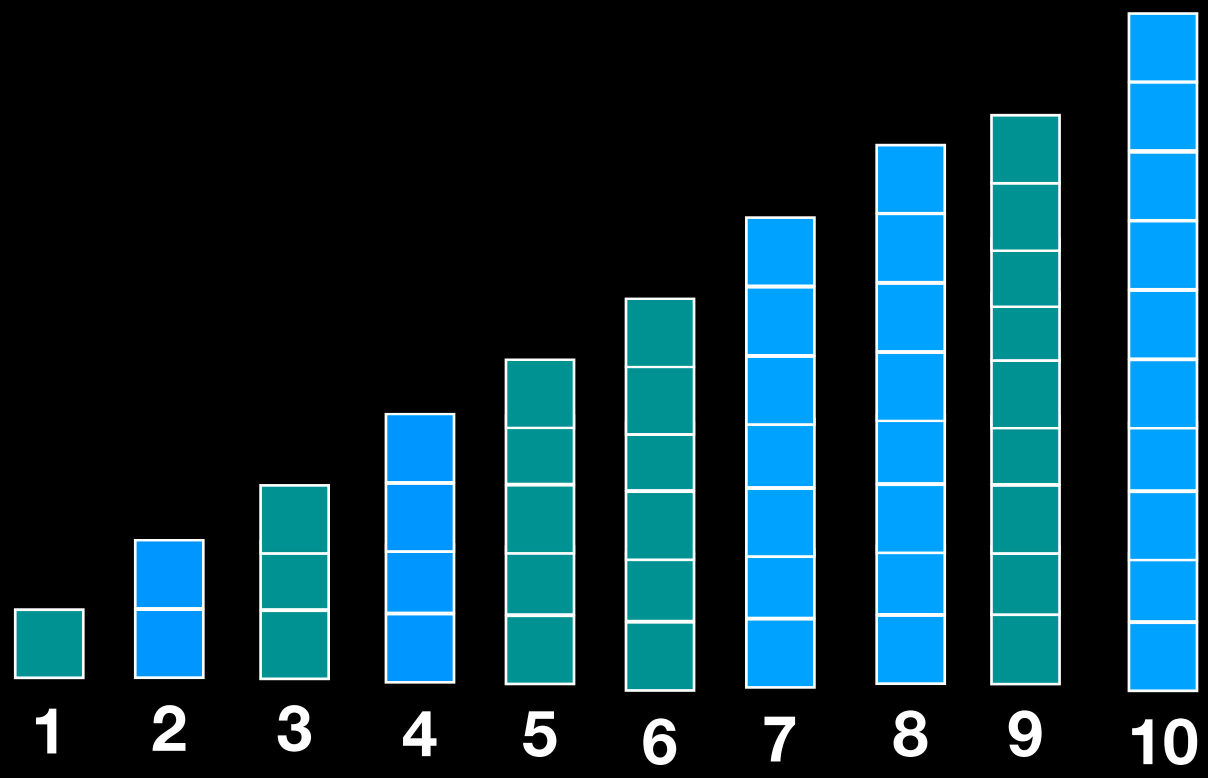
Key Insight: Merge is linear



Key Insight: Merge is linear

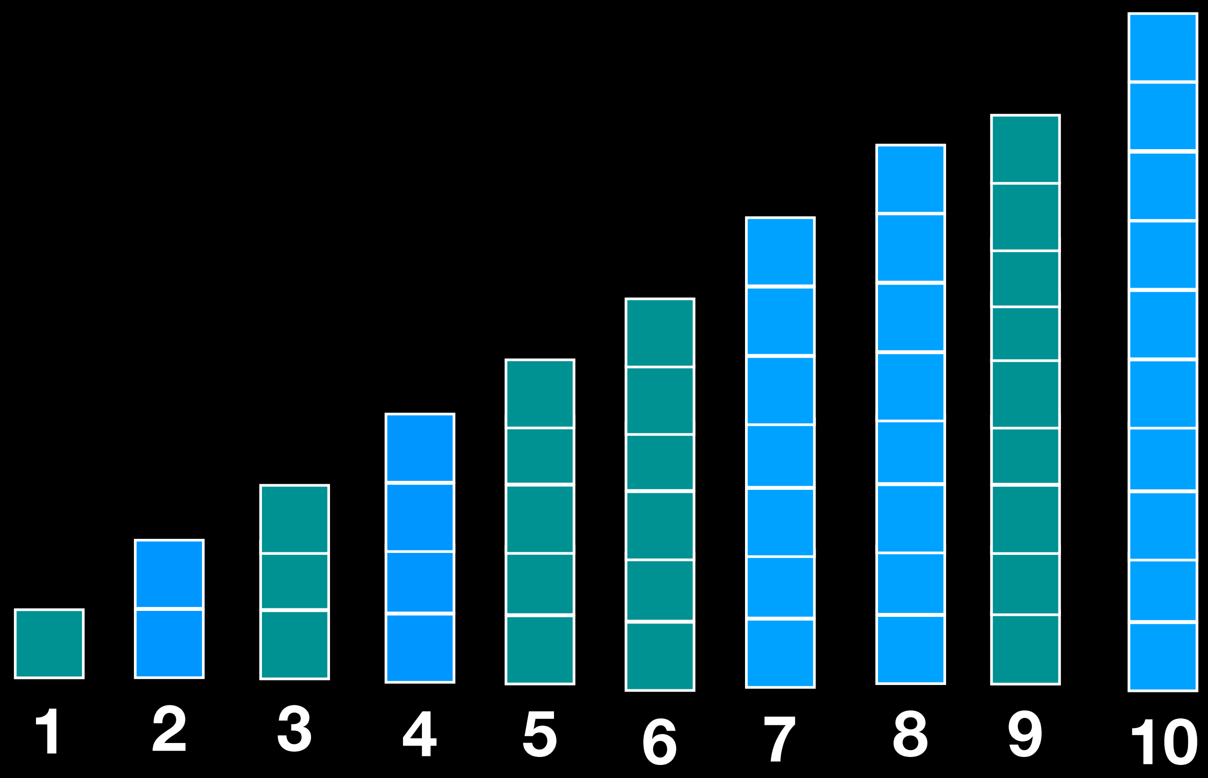


Key Insight: Merge is linear



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

11	64	158	195	260	599	932

$T(^1/_2 n) \approx {}^1/_4\, T(n)$

$T(^1/_2 n) \approx {}^1/_4\, T(n)$

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

$T(^1/_2 n) \approx {}^1/_4\, T(n)$ $T(^1/_2 n) \approx {}^1/_4\, T(n)$

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]

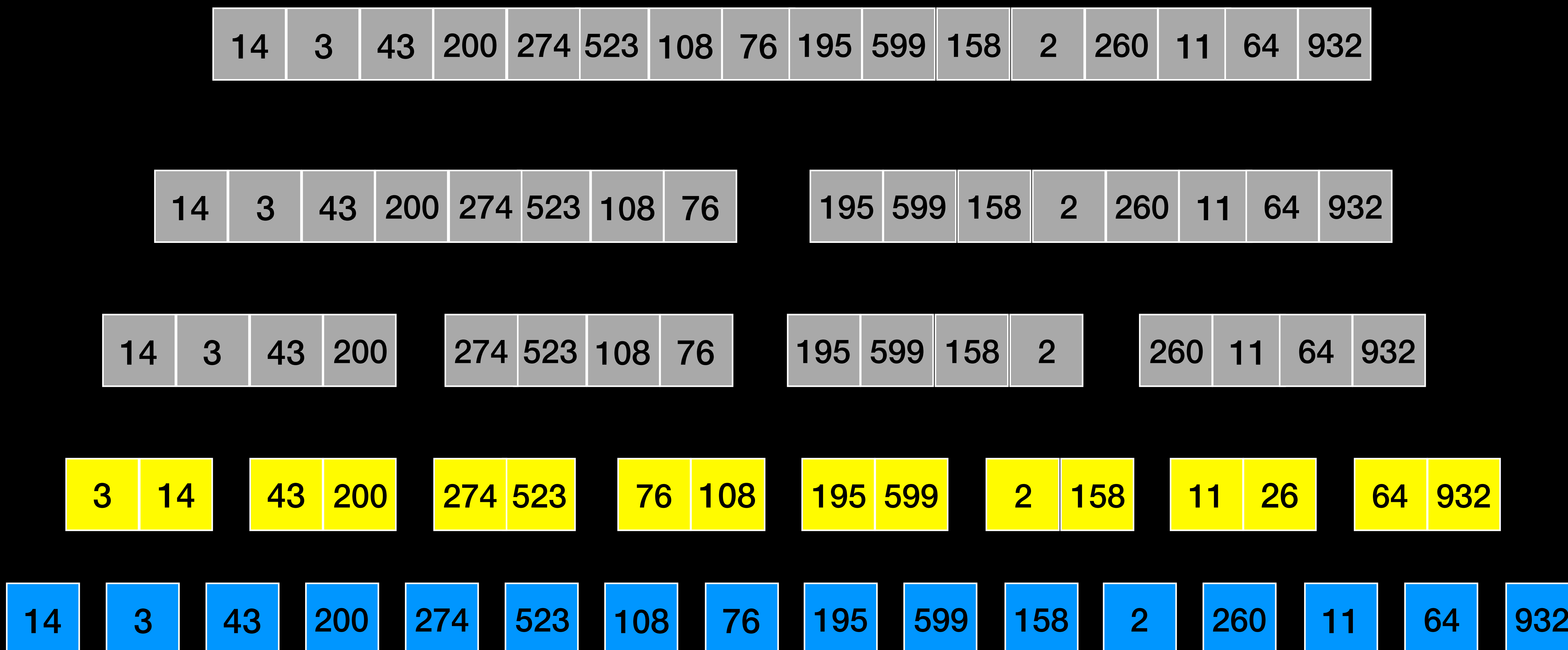
[14, 3] [43, 200] [274, 523] [108, 76] [195, 599] [158, 2] [260, 11] [64, 932]

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

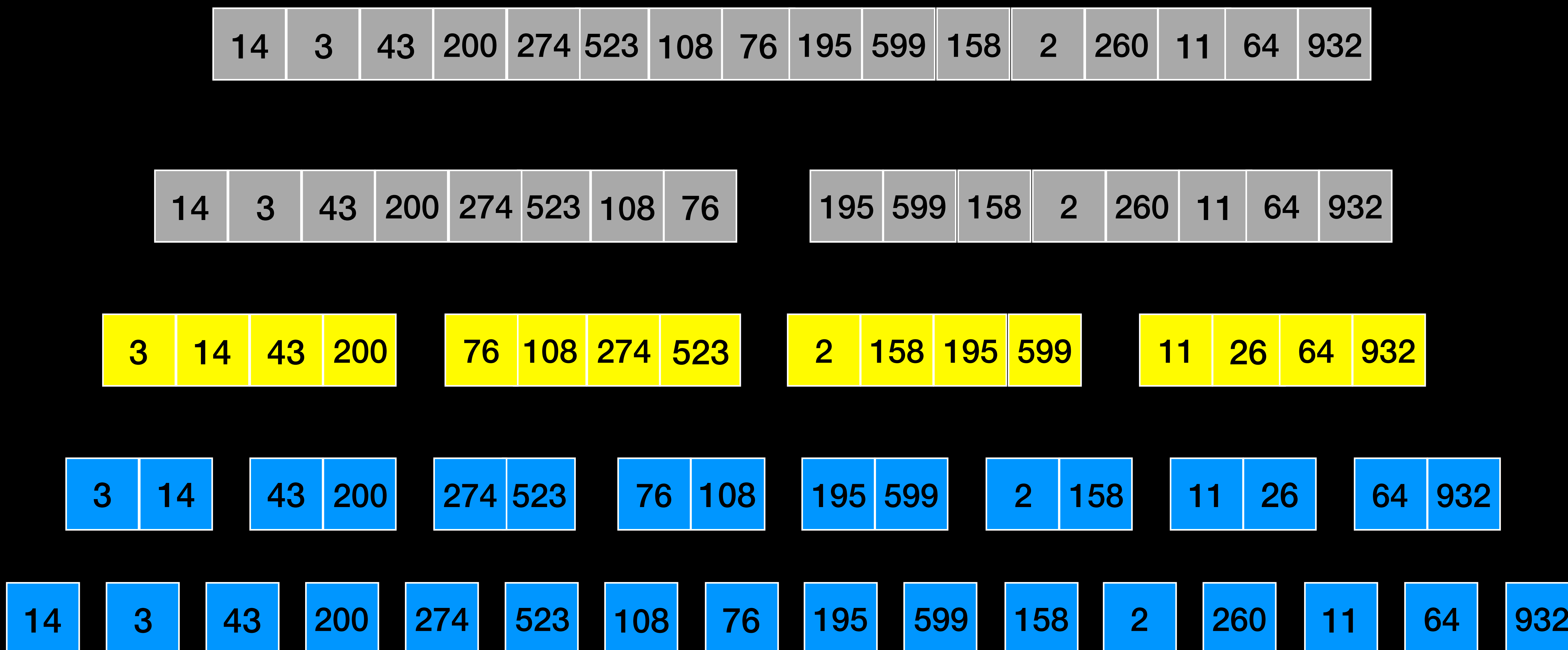
Merge Sort


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

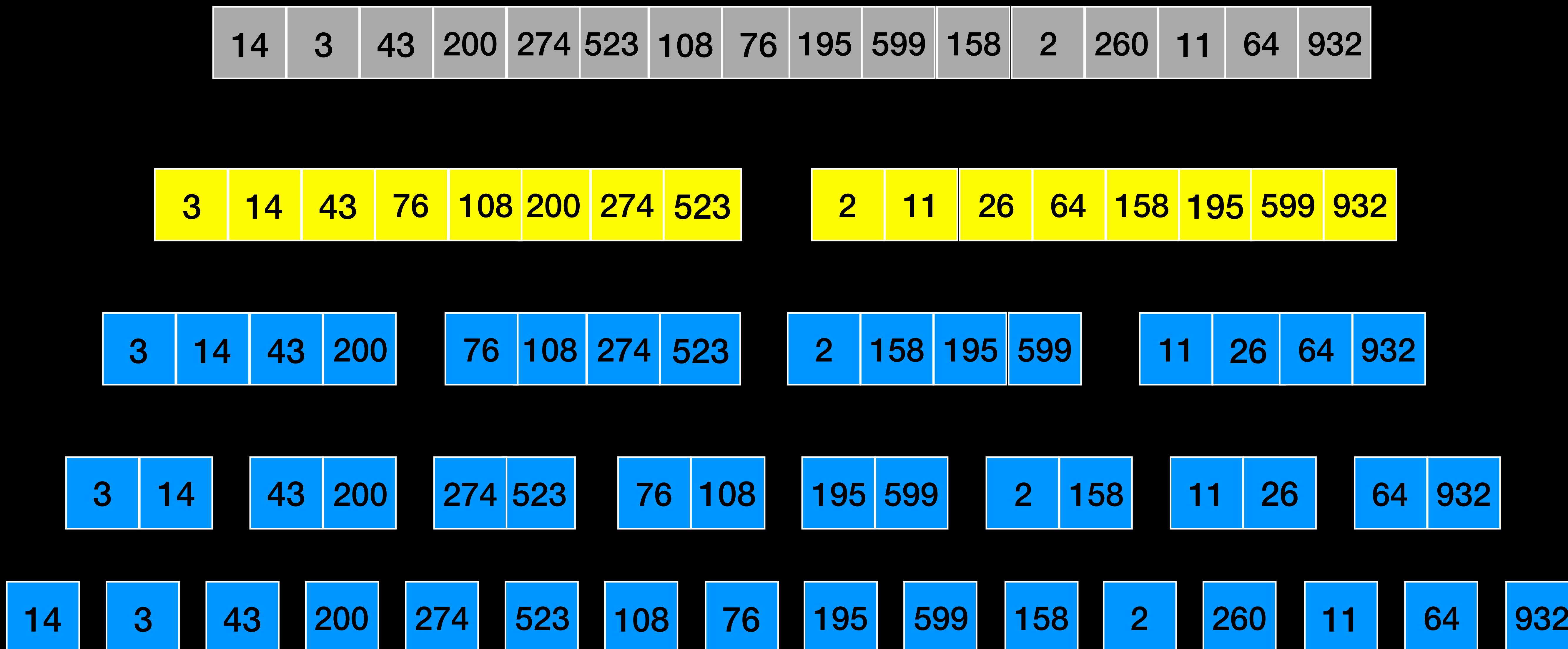
Merge Sort


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

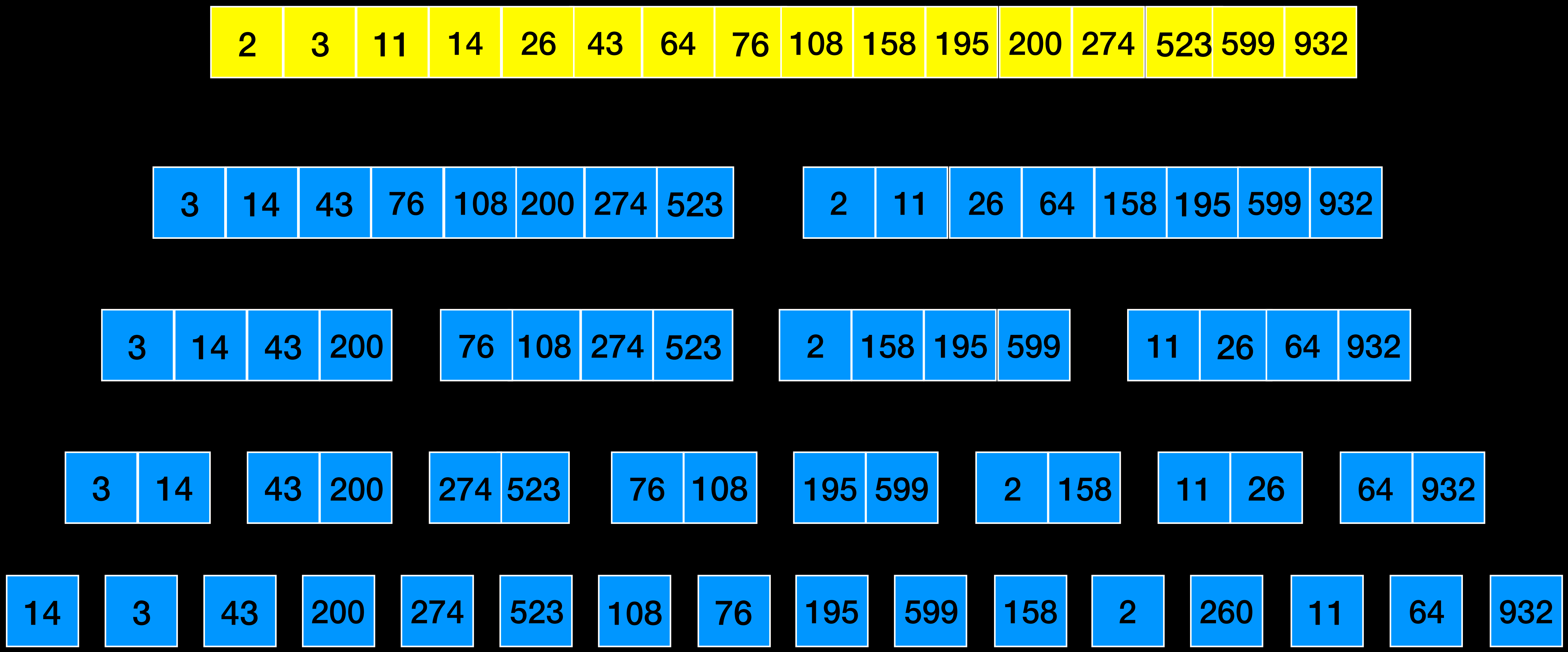

14 3 43 200 274 523 108 76 195 599 158 2 260 11 64 932
14 3 43 200 274 523 108 76
195 599 158 2 260 11 64 932
3 14 43 200
76 108 274 523
2 158 195 599
11 26 64 932
3 14
43 200
274 523
76 108
195 599
2 158
11 26
64 932
14 3 43 200 274 523 108 76 195 599 158 2 260 11 64 932

Merge Sort



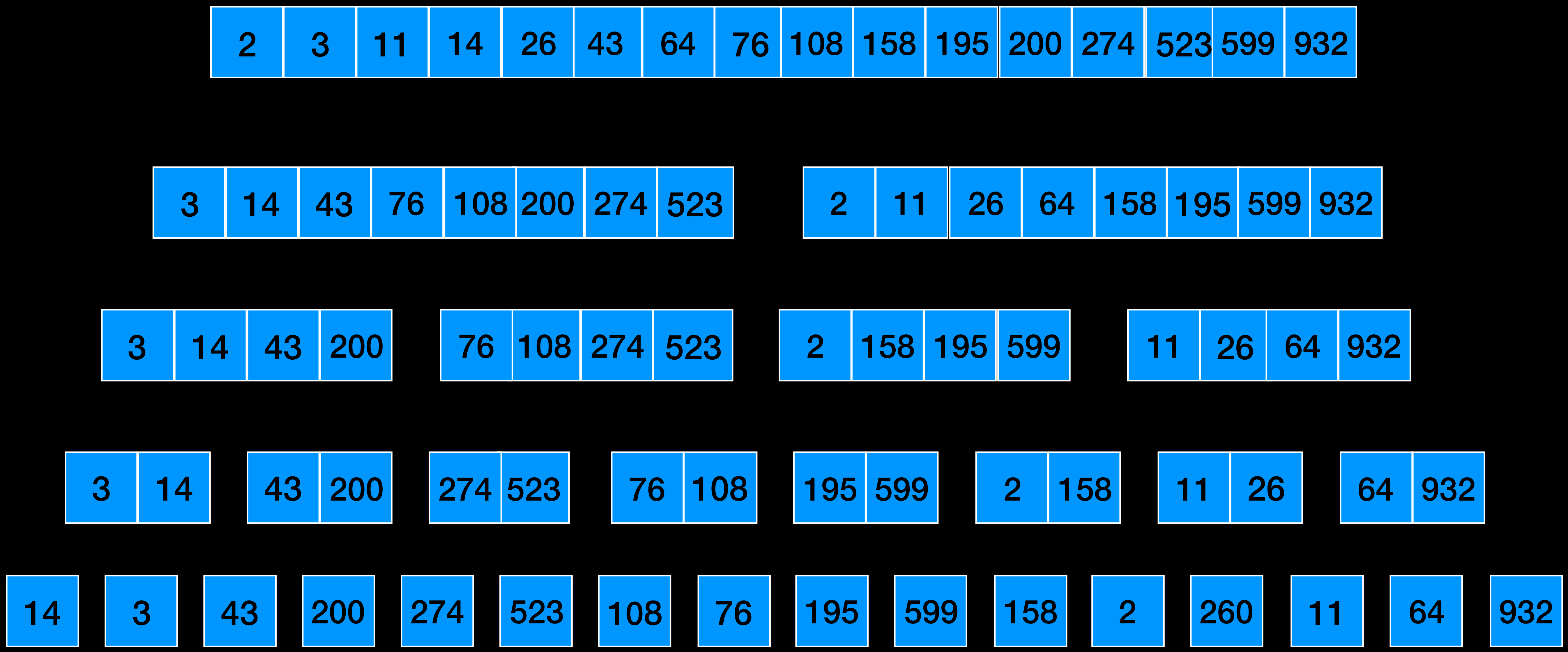
14 3 43 200 274 523 108 76 195 599 158 2 260 11 64 932
3 14 43 76 108 200 274 523
2 11 26 64 158 195 599 932
3 14 43 200
76 108 274 523
2 158 195 599
11 26 64 932
3 14
43 200
274 523
76 108
195 599
2 158
11 26
64 932
14
3
43
200
274
523
108
76
195
599
158
2
260
11
64
932

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)

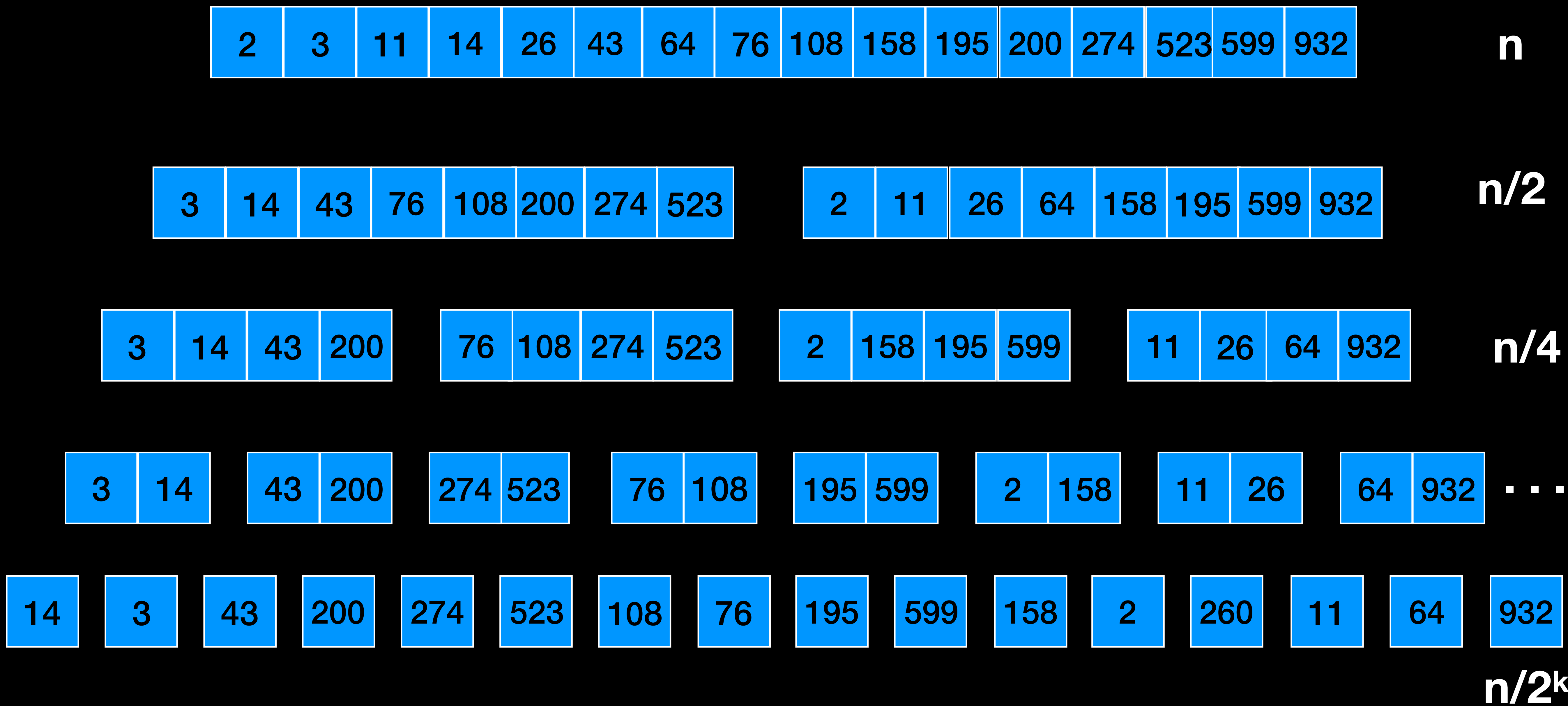
3 14 43 200 | 76 108 274 523 | 2 158 195 599 | 11 26 64 932 O(n)

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

O(n)

Merge Sort Analysis



Merge 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	**. . .**

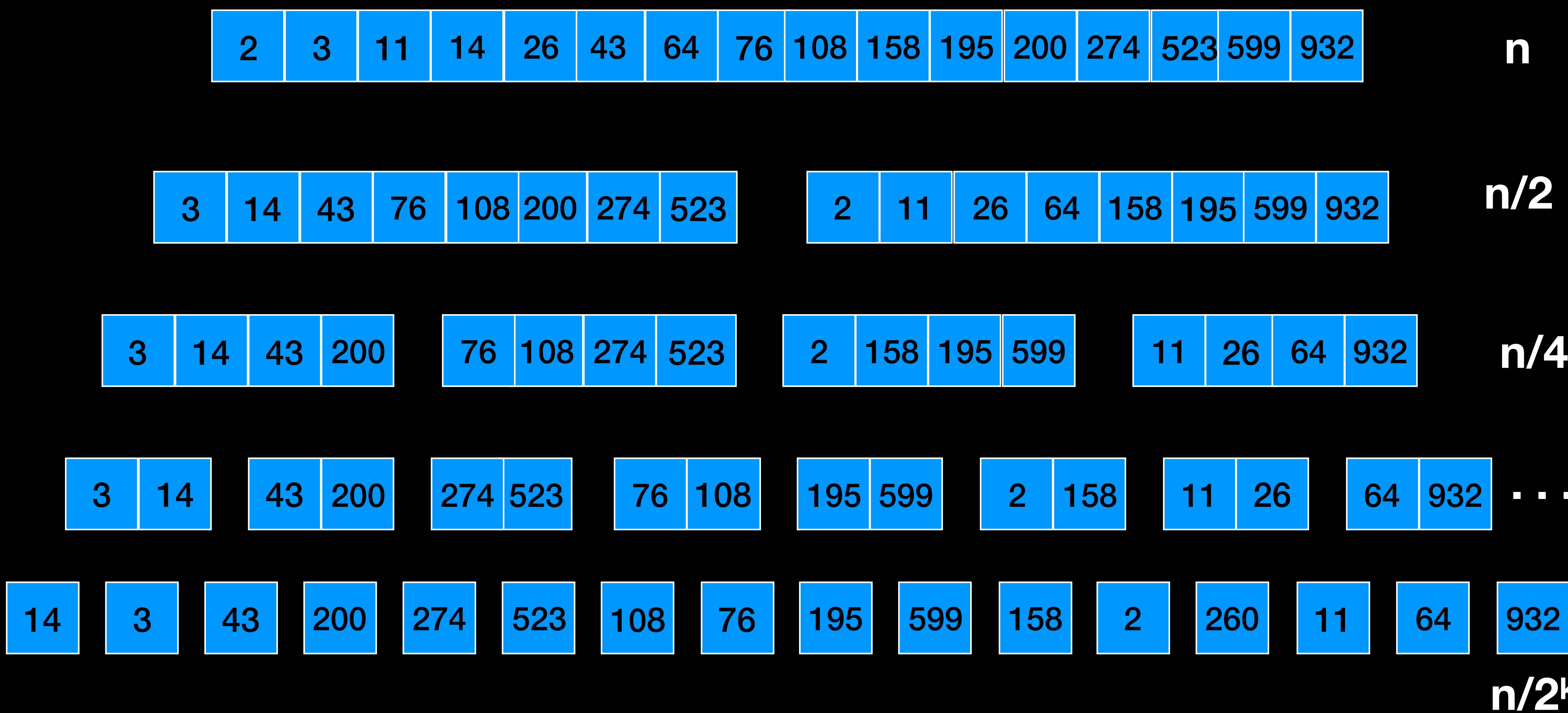
14	3	43	200	274	523	108	76	195	599	158	2	260	11	64	932	**$n/2^k$**

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

$n = 2^k$

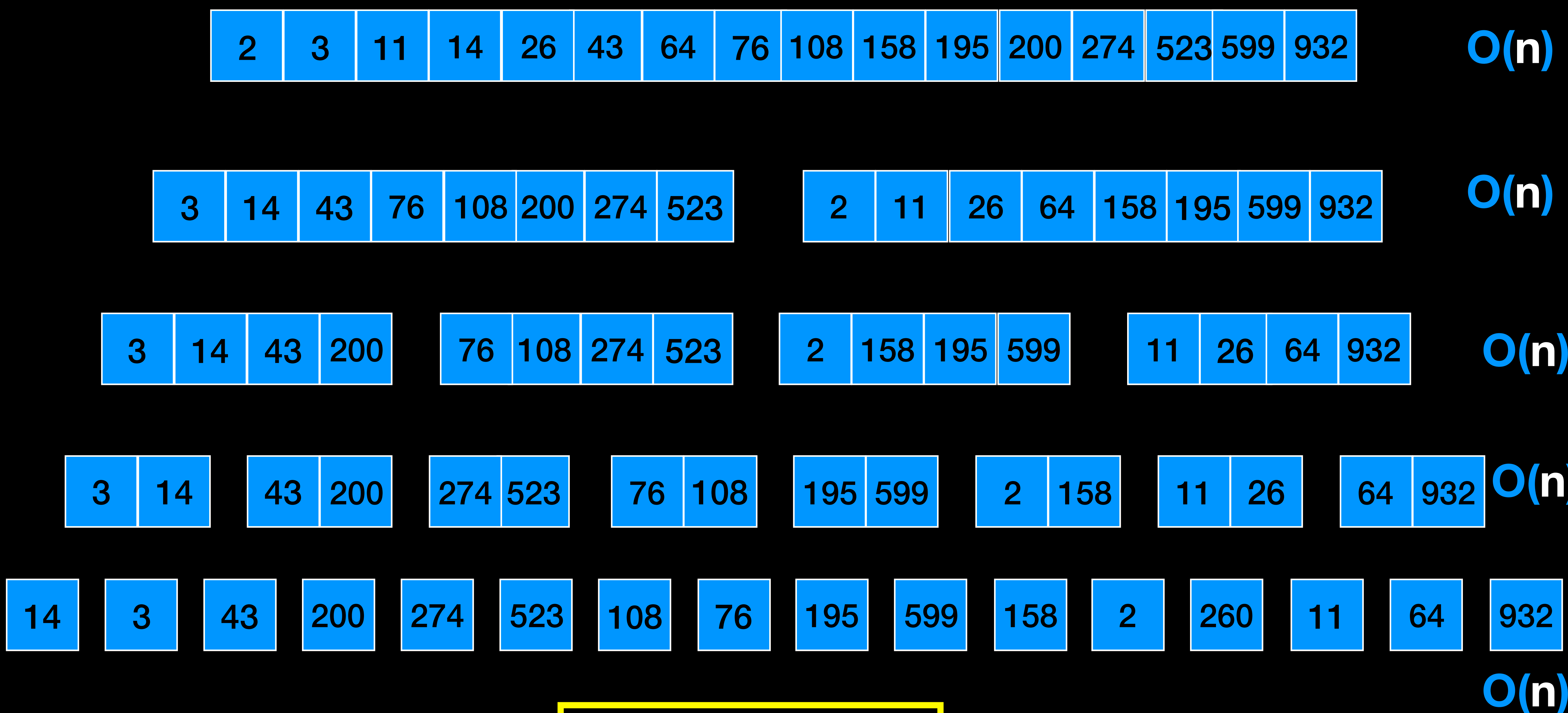
$\log_2 n = k$

Merge Sort Analysis



Merge $\log_2 n$ times

Merge Sort Analysis



$\boxed{O(n \log n)}$

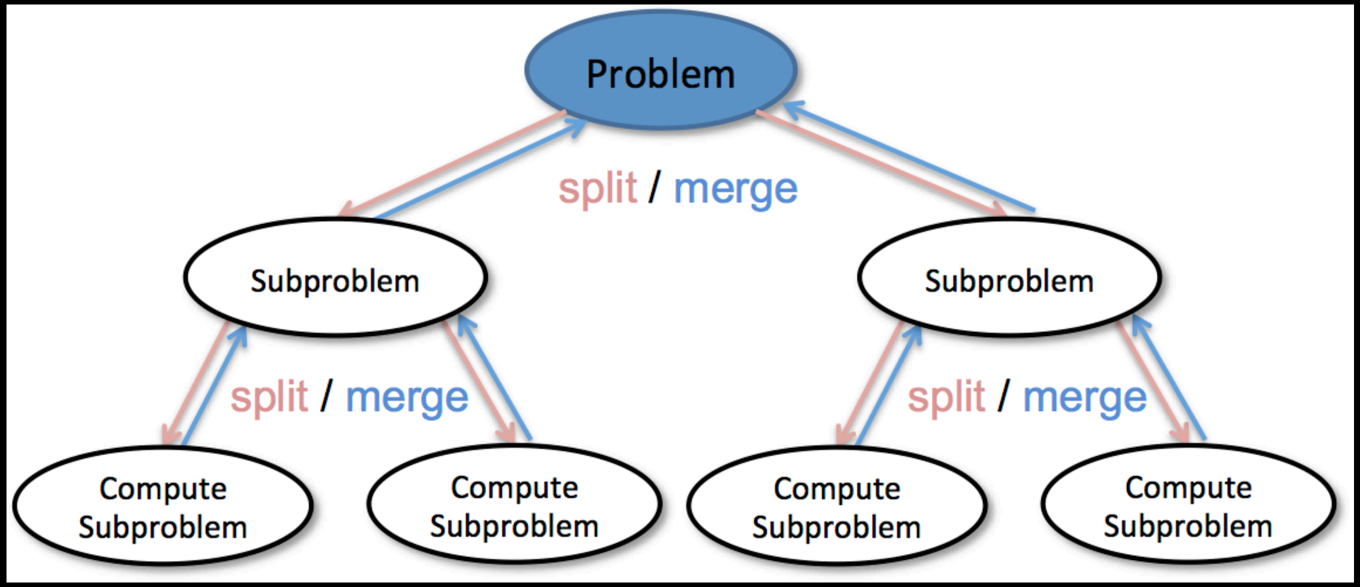
Merge Sort

How would you code this up?

Merge Sort

How would you code this up?

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, sorted_array)
}
```

Merge Sort Analysis

Execution time does NOT depend on initial arrangement of data

$O(n \log n)$ comparisons and data moves

$\Omega(n \log n)$

Stable

Best we can do with *comparison-based* sorting in the worst case
=> can't beat $O(n \log n)$

Space overhead: auxiliary array at each merge step

What we have so far

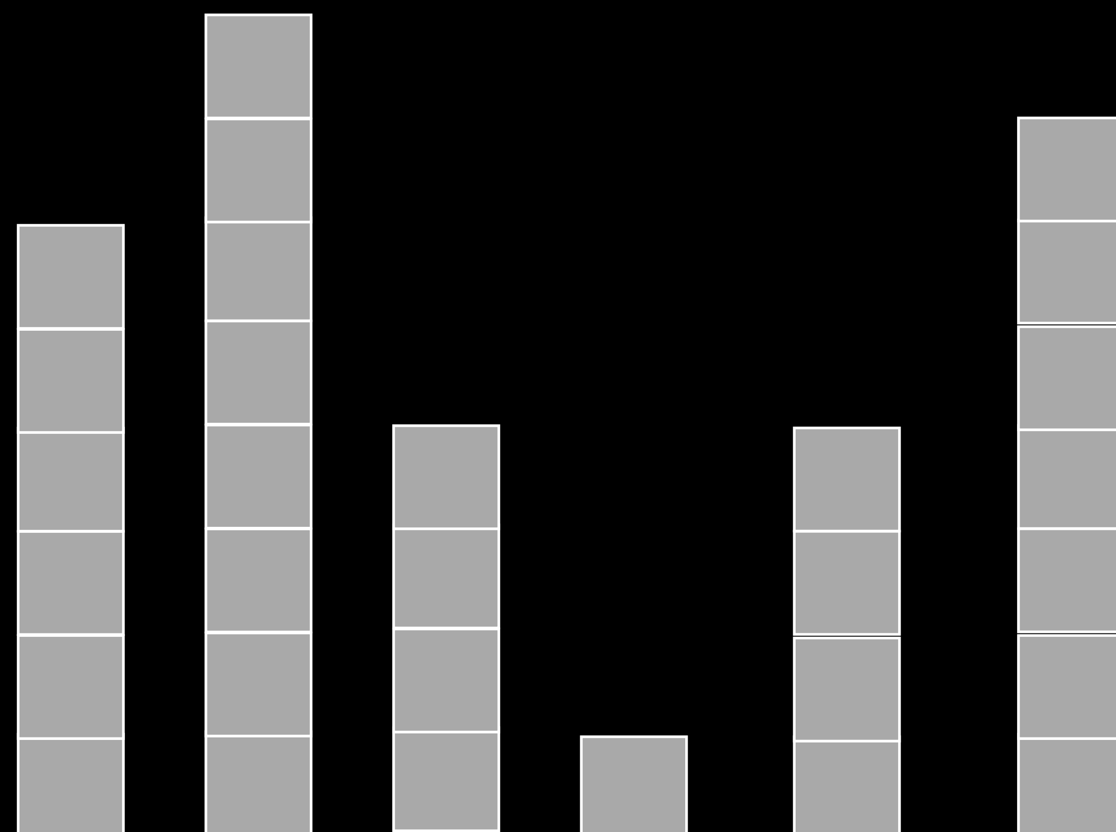
	O	Ω
Selection Sort	$O(n^2)$	$\Omega(n^2)$
Insertion Sort	$O(n^2)$	$\Omega(n)$
Bubble Sort	$O(n^2)$	$\Omega(n)$
Merge Sort	$O(n \log n)$	$\Omega(n \log n)$

Quick Sort

Unsorted

Sorted

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

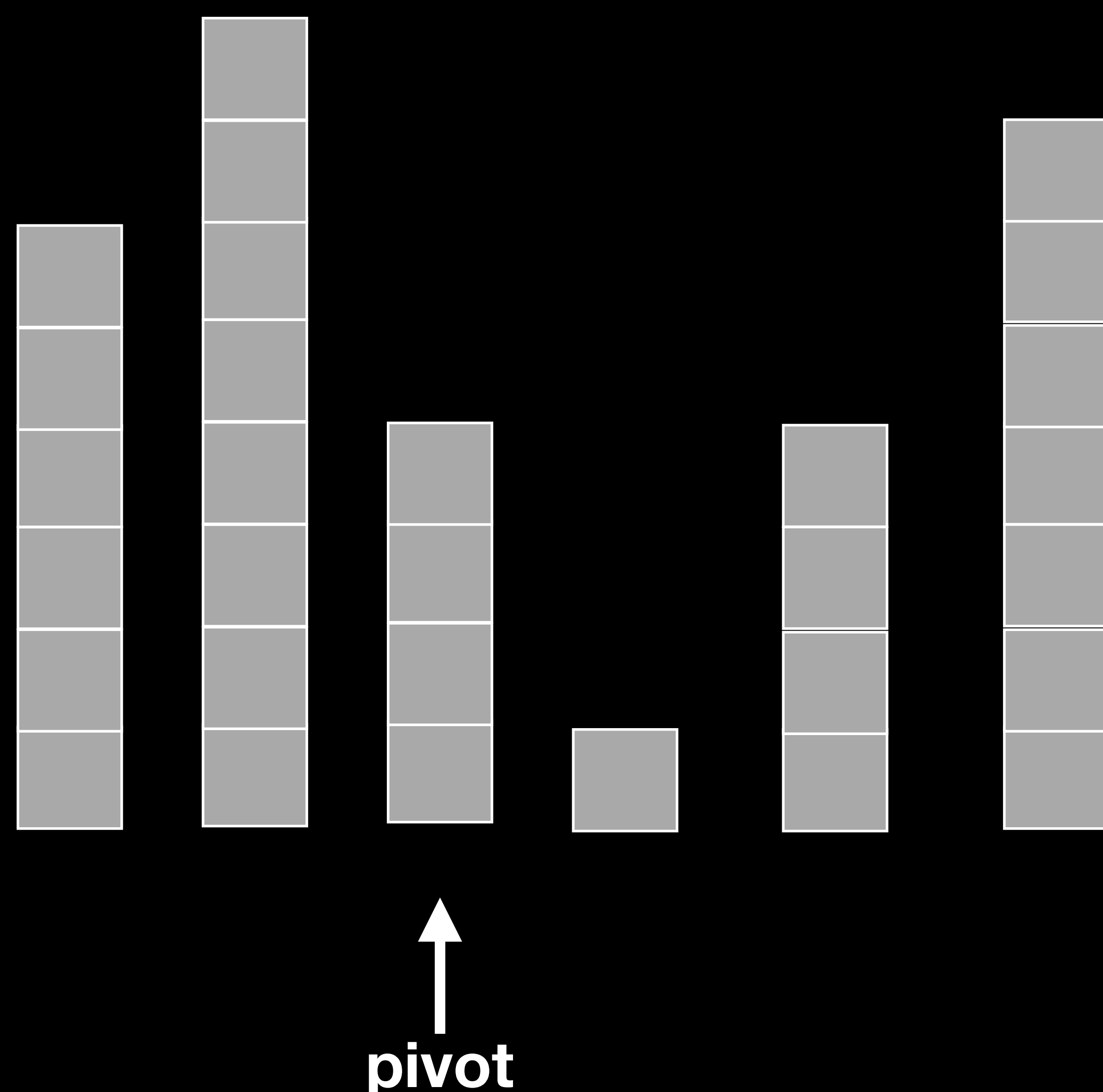


Quick Sort

Unsorted

Sorted

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

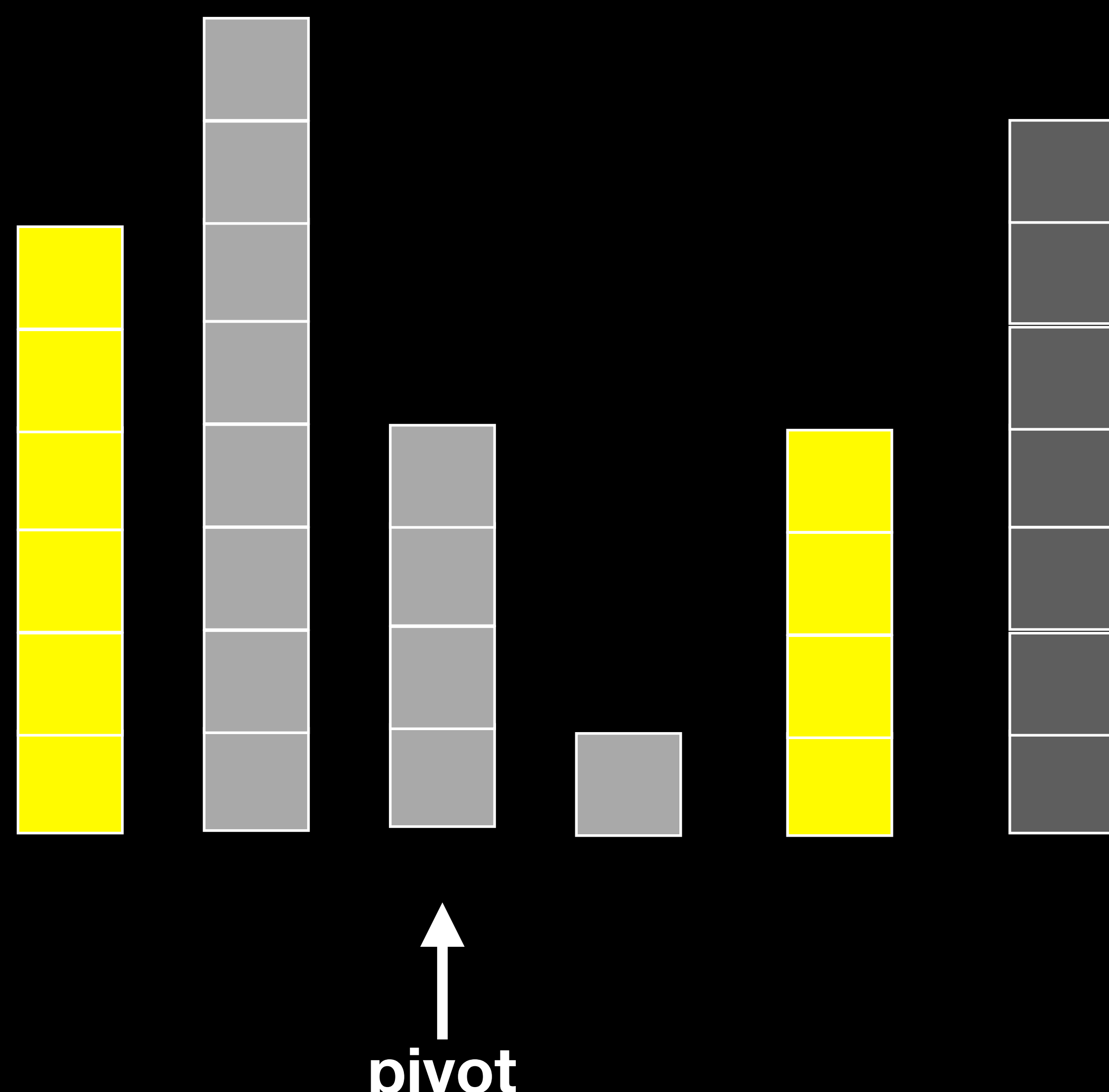


Quick Sort

Unsorted

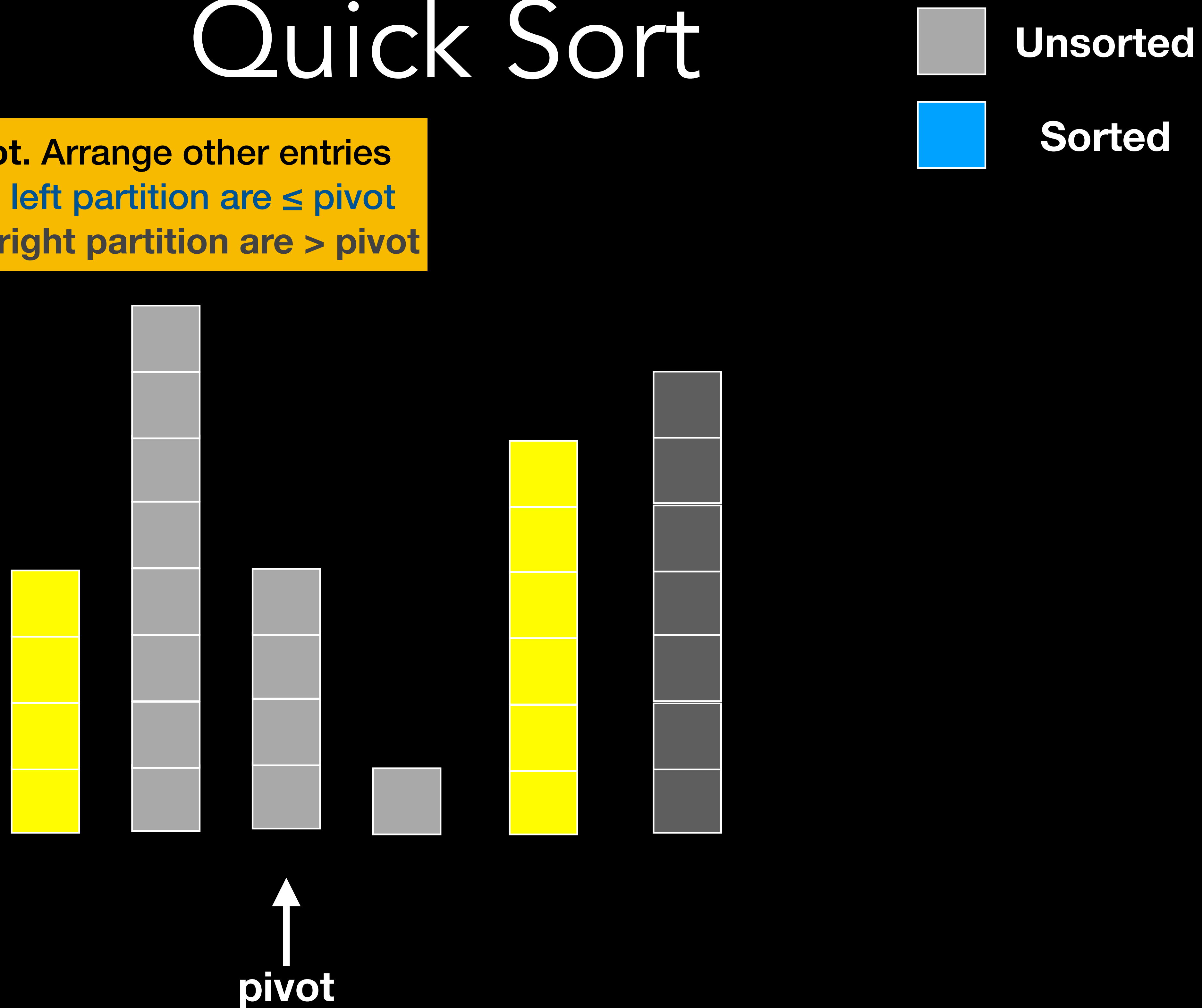
Sorted

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



Quick Sort

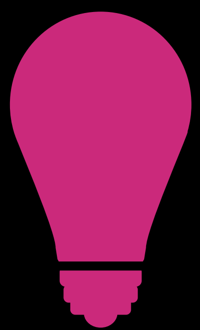
Select a **pivot.** Arrange other entries s.t. entries in left partition are ≤ pivot and entries in right partition are > pivot



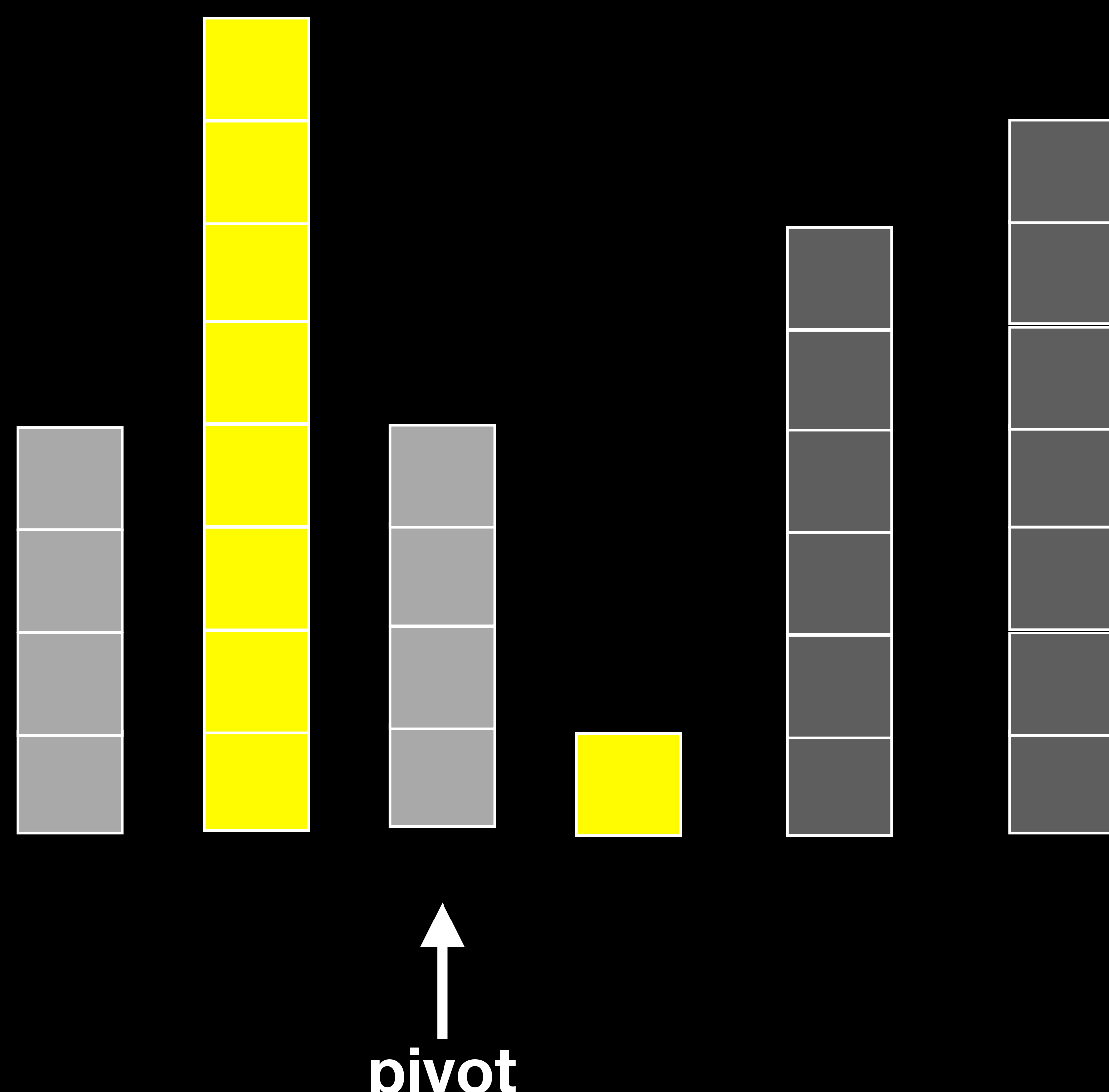
Quick Sort

Unsorted

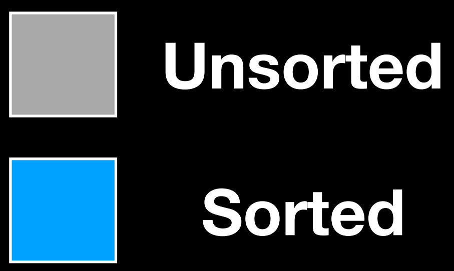
Sorted



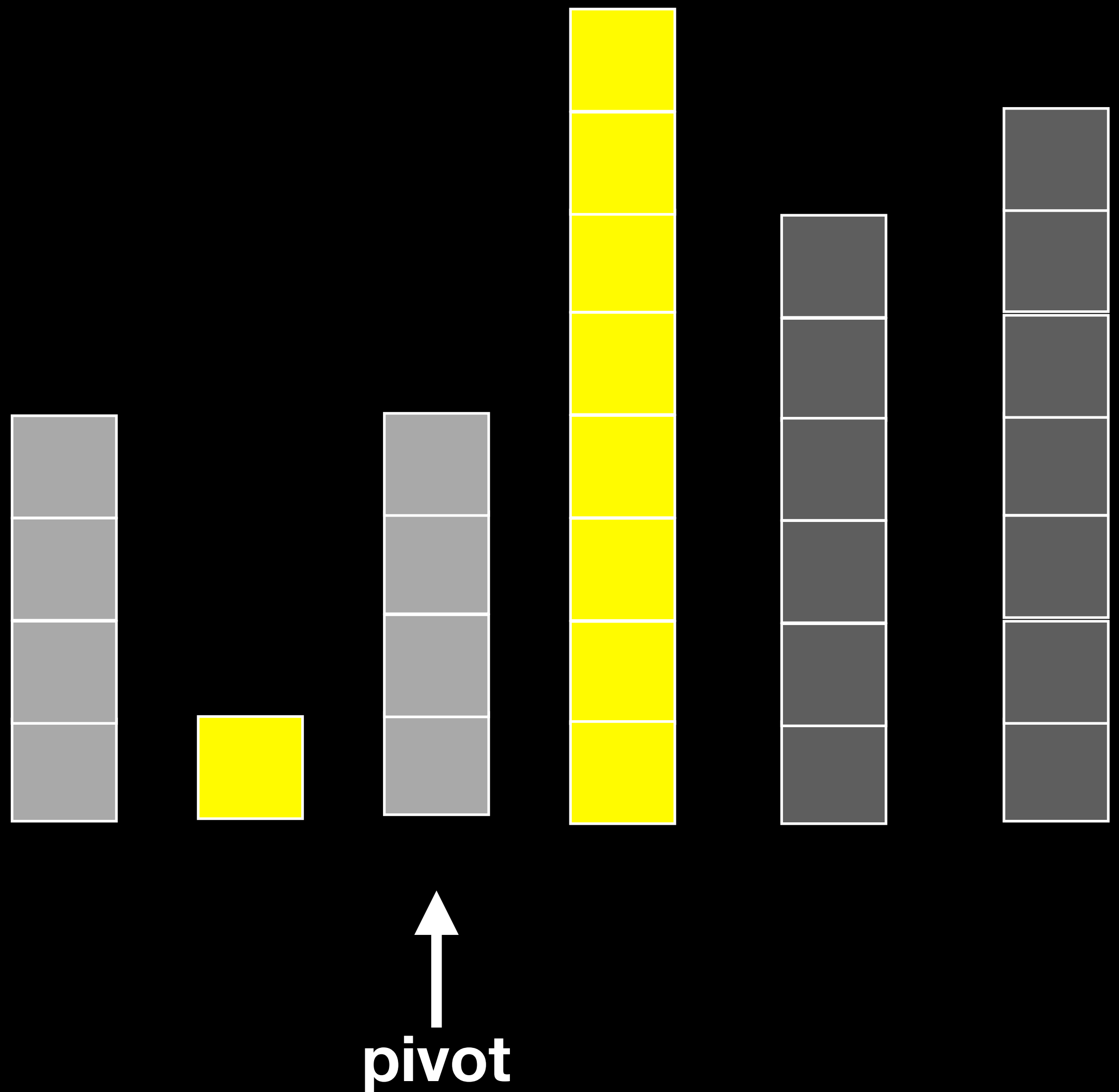
Select a **pivot.** Arrange other entries s.t. entries in left partition are ≤ pivot and entries in right partition are > pivot



Quick Sort



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

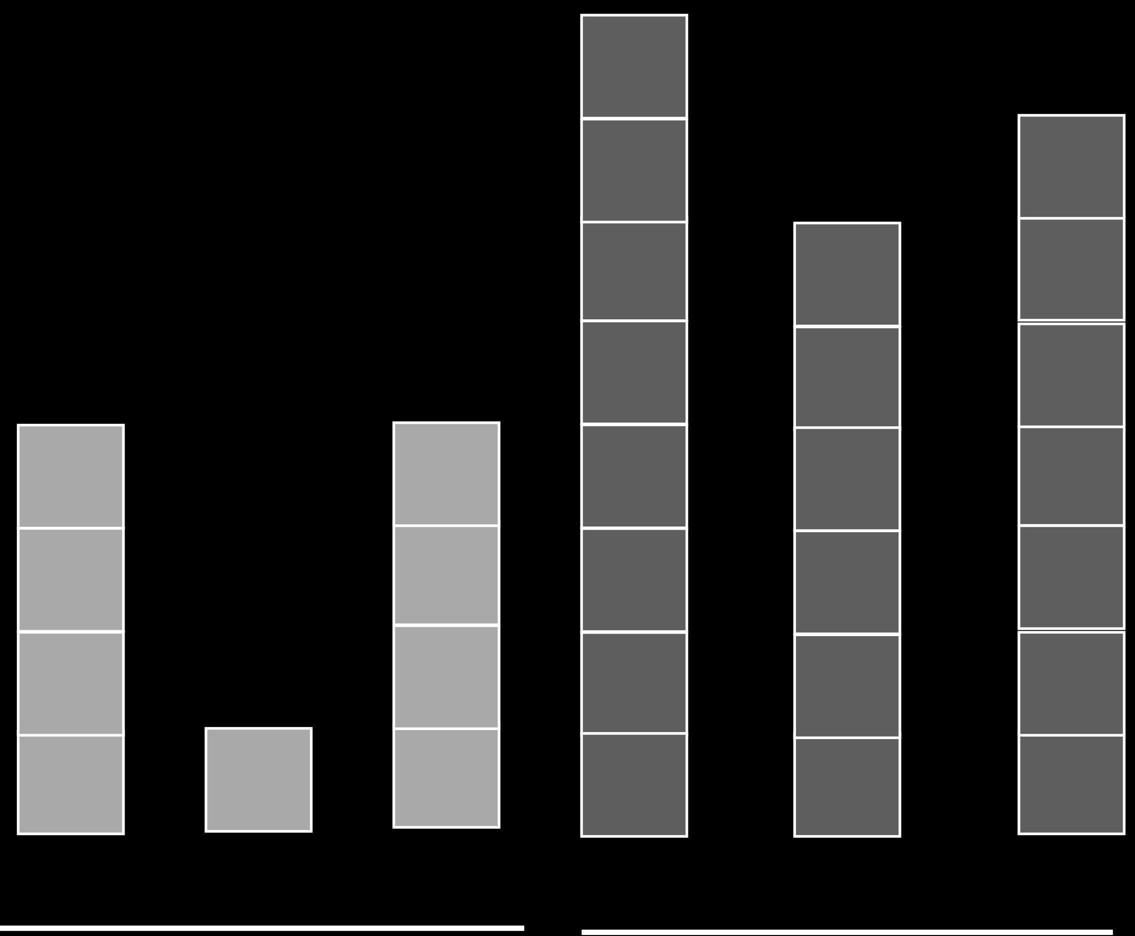


Quick Sort

Unsorted

Sorted

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



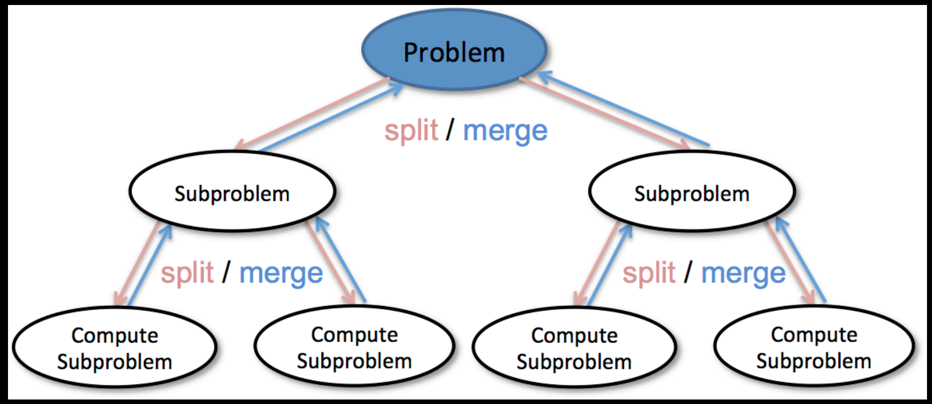
≤ pivot

`quickSort( )`

> pivot

`quickSort( )`

Quick Sort Analysis



Divide and Conquer

n comparisons for each partition

How many subproblems? => Depends on pivot selection

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

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

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

95								6								13

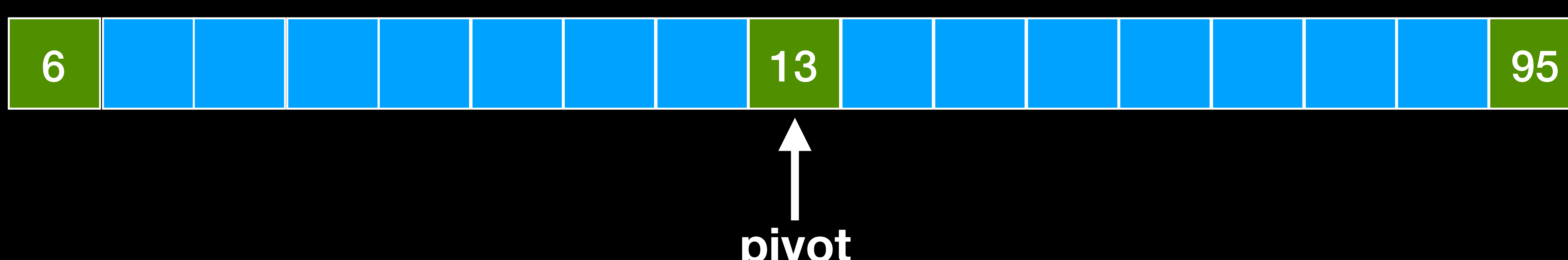
How to select pivot?

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

Implementation tweaks (e.g. smart pivot selection, speed up base case) can improve actual runtime

$O(n^2)$ comparisons and data moves

$\Omega(n \log n)$

Unstable

Quick Sort

```
void quickSort(array, first, last)
{
    if last - first + 1 = MIN_SIZE
         //base case with improvement
        insertionSort(array, first, last)
    else
        pivot_index = partition(array, first, last)
        quickSort(array, first, pivot_index - 1)
        quickSort(array, pivot_index + 1, last)

}
```

What we have so far

	O	Ω
Selection Sort	$O(n^2)$	$\Omega(n^2)$
Insertion Sort	$O(n^2)$	$\Omega(n)$
Bubble Sort	$O(n^2)$	$\Omega(n)$
Merge Sort	$O(n \log n)$	$\Omega(n \log n)$
Quick Sort	$O(n^2)$	$\Omega(n \log n)$