

# Algorithmic Analysis and Sorting

## Part Two

Recap from Last Time

# Big-O Notation

- **Big-O notation** is a quantitative way to describe the runtime of a piece of code.
- For example, the runtime of this code snippet is  **$O(n)$** , where  $n$  is the size of the vector:

```
for (int i = 0; i < vec.size(); i++) {  
    cout << vec[i] << endl;  
}
```

# Big-O Notation

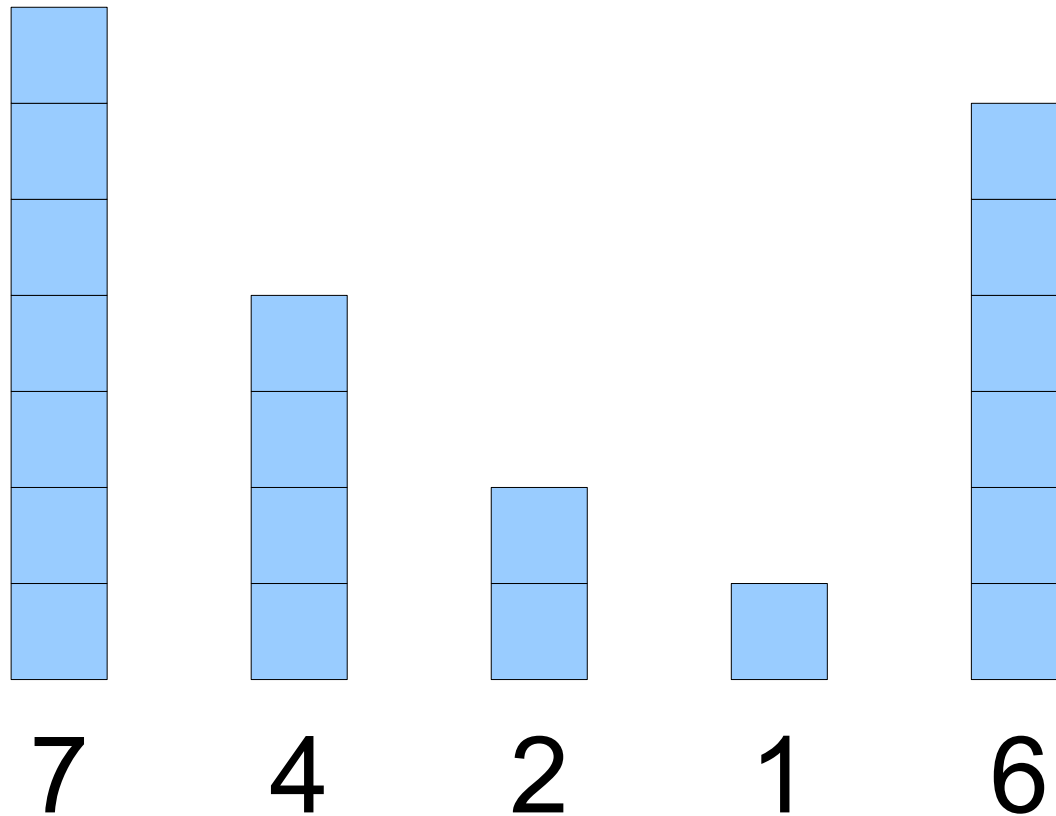
- **Big-O notation** is a quantitative way to describe the runtime of a piece of code.
- For example, the runtime of this code snippet is  **$O(n^2)$** , where  $n$  is the size of the vector:

```
for (int i = 0; i < vec.size(); i++) {  
    for (int j = 0; j < vec.size(); j++) {  
        cout << (vec[i] + vec[j]) << endl;  
    }  
}
```

# Sorting Algorithms

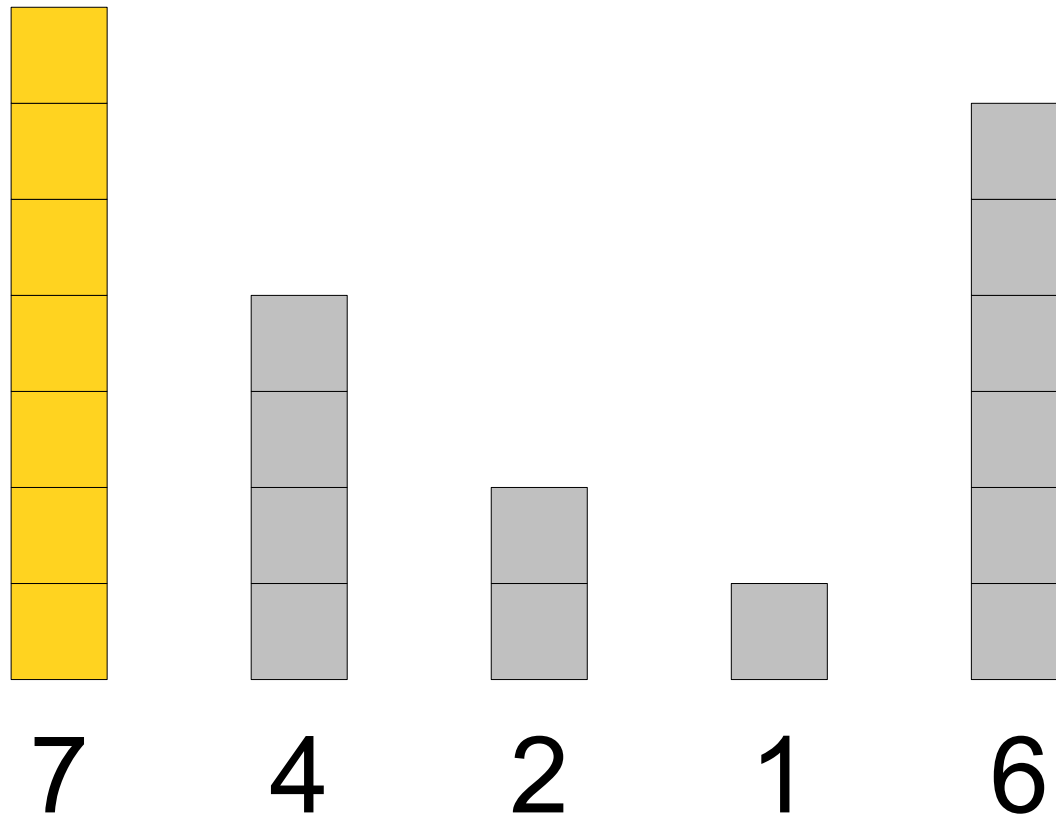
- The ***sorting problem*** is to take in a list of things (integers, strings, etc.) and rearrange them into sorted order.
- Last time, we saw ***insertion sort***, an algorithm that runs in time  $O(n^2)$ .

# An Initial Idea: *Insertion Sort*



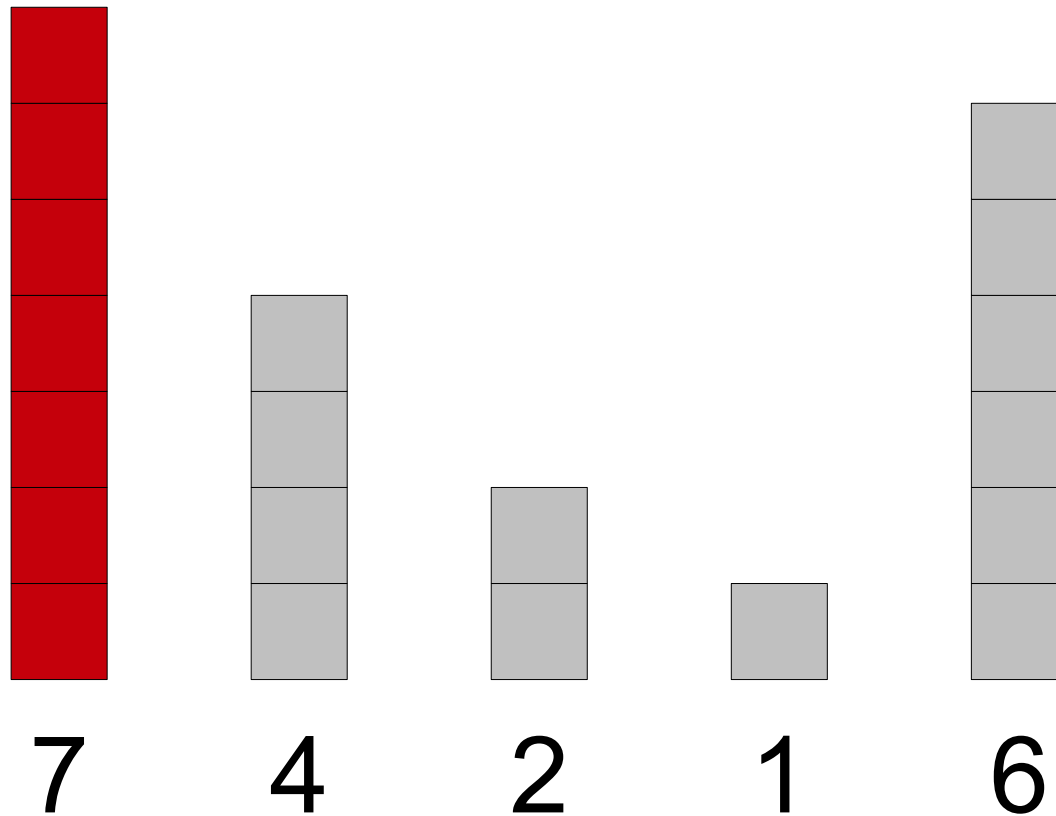
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

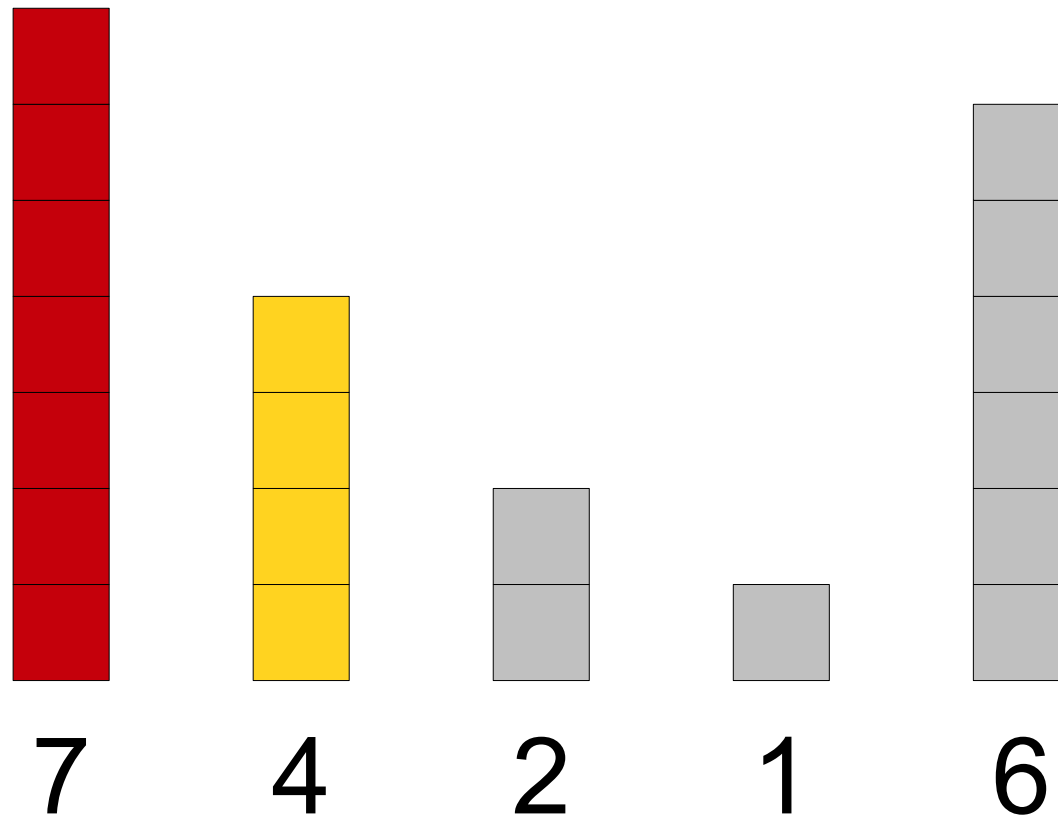
# An Initial Idea: *Insertion Sort*



**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

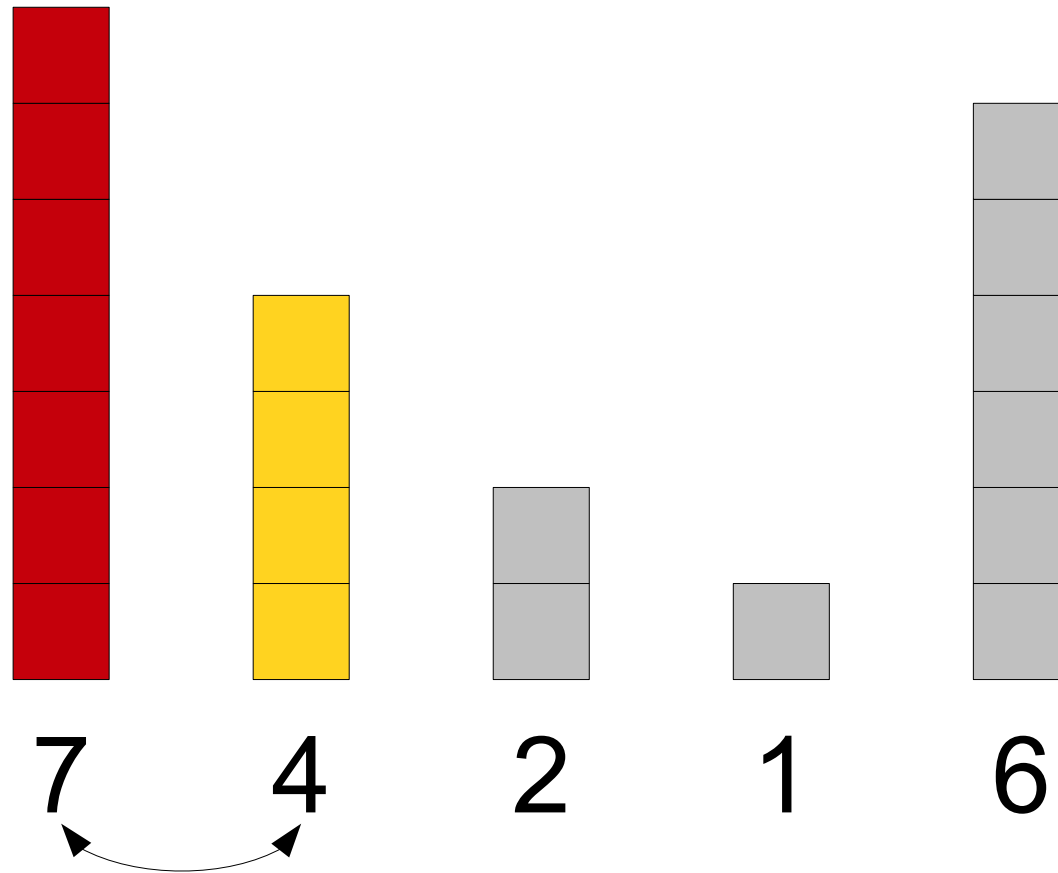


# An Initial Idea: *Insertion Sort*



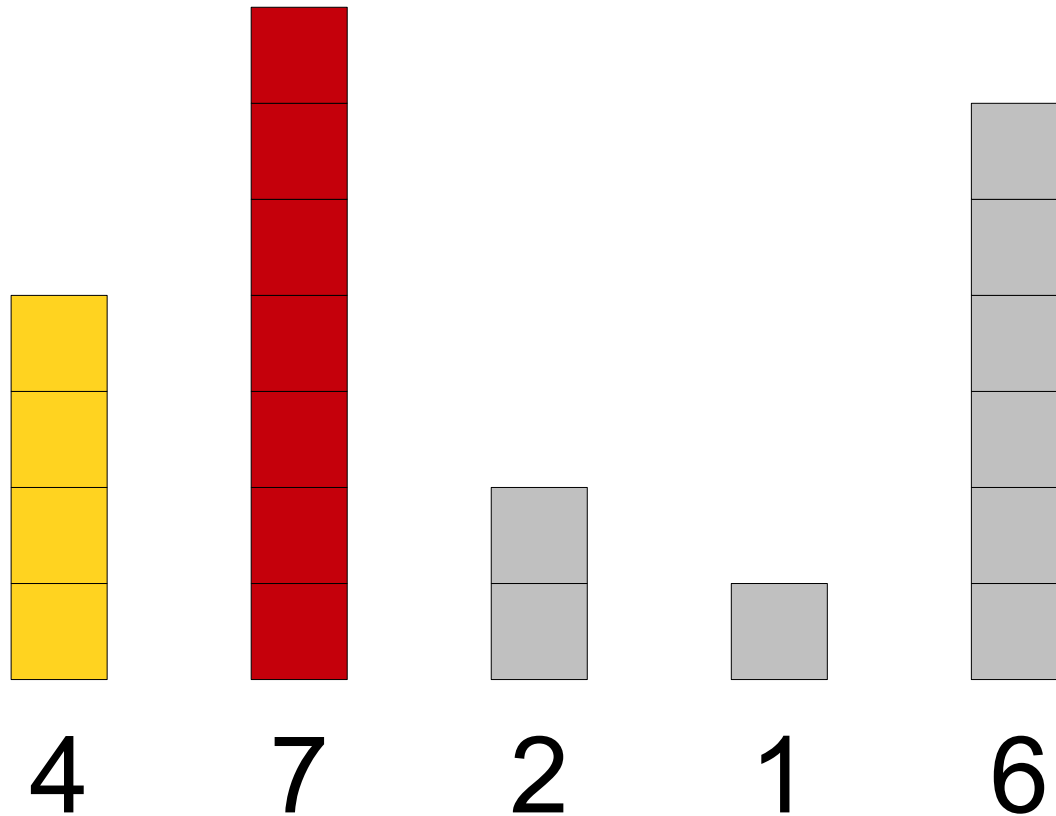
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



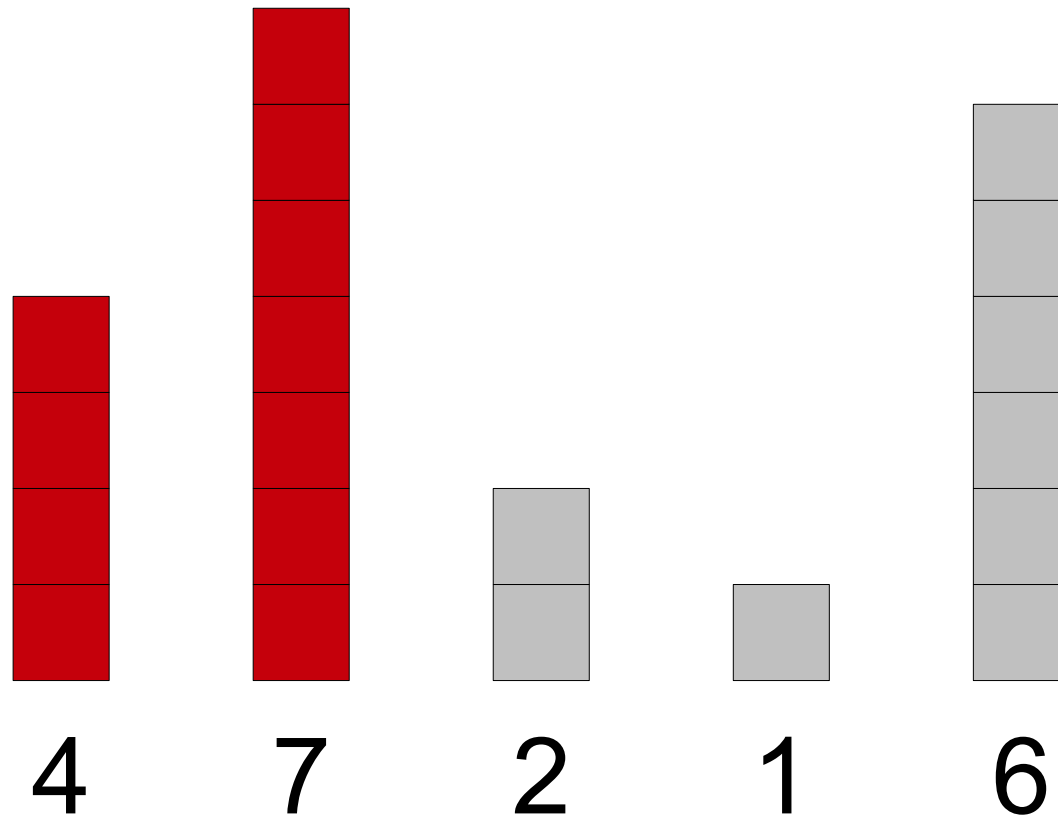
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



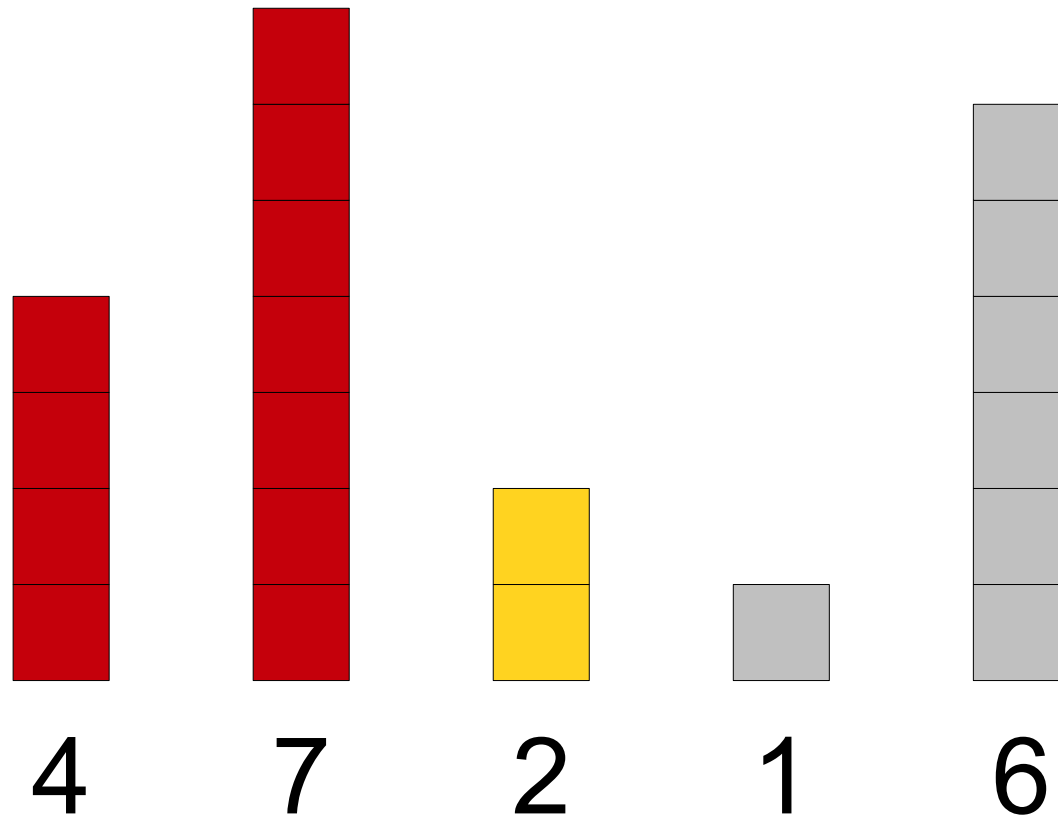
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



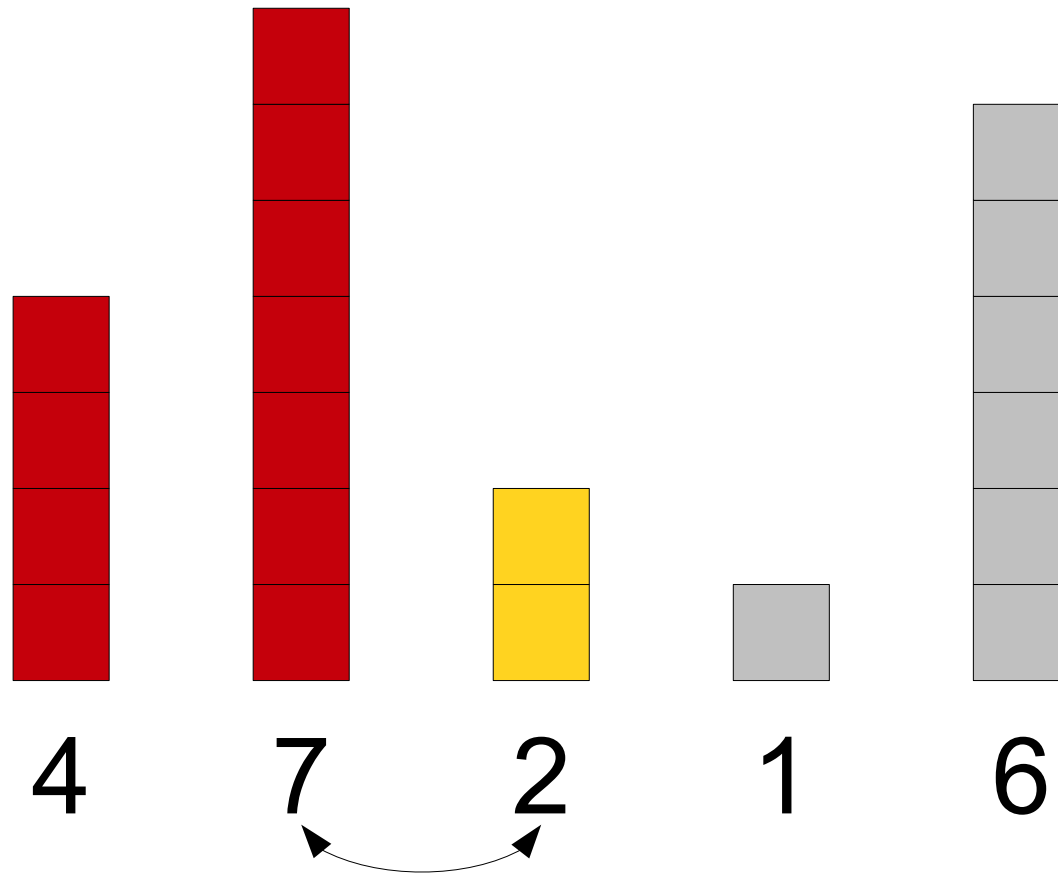
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



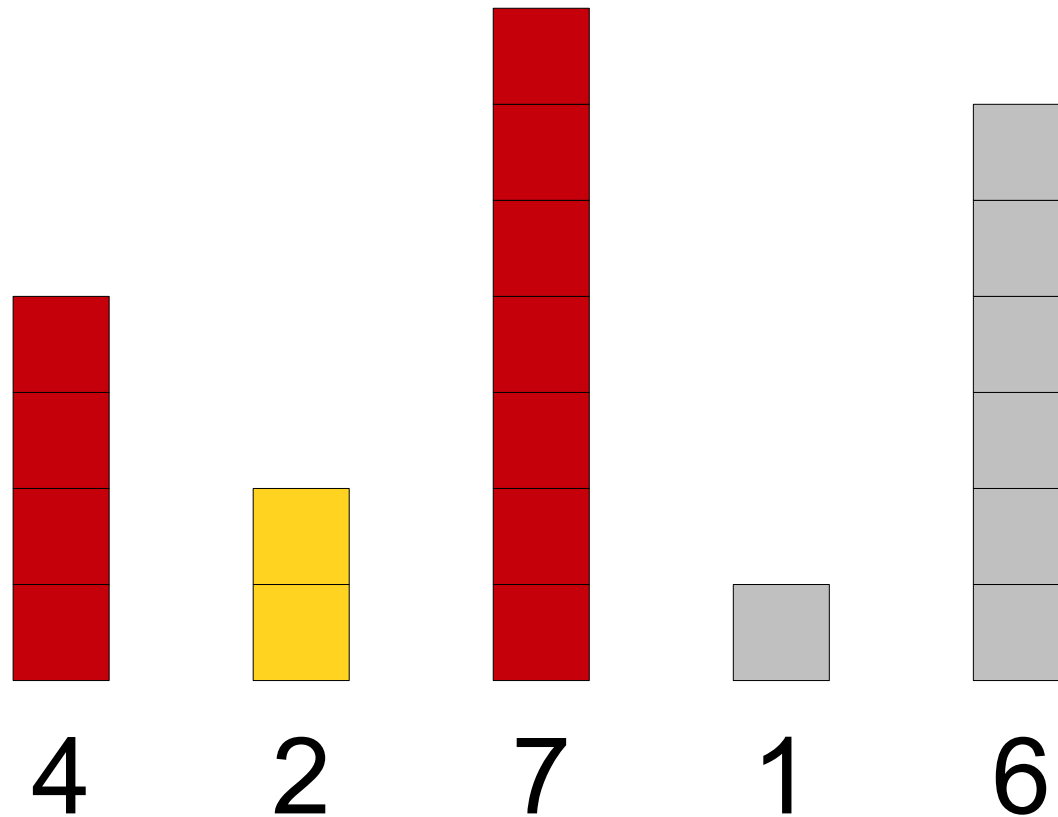
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



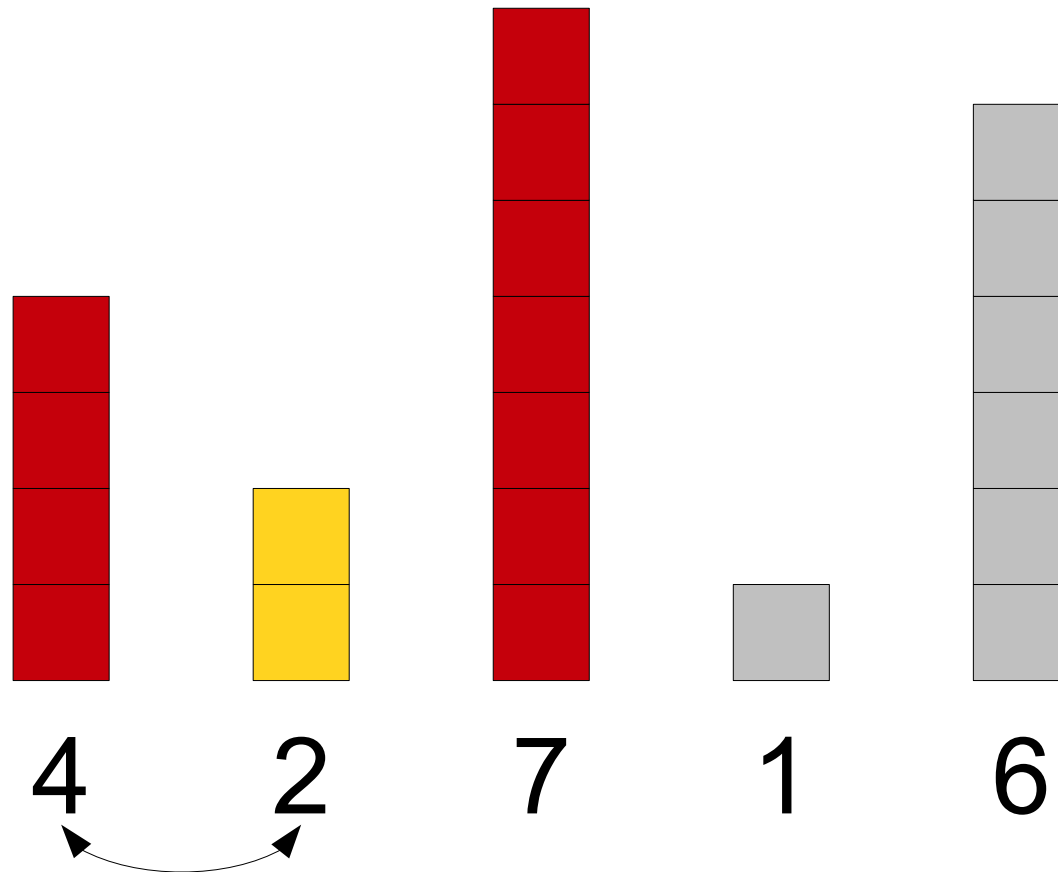
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

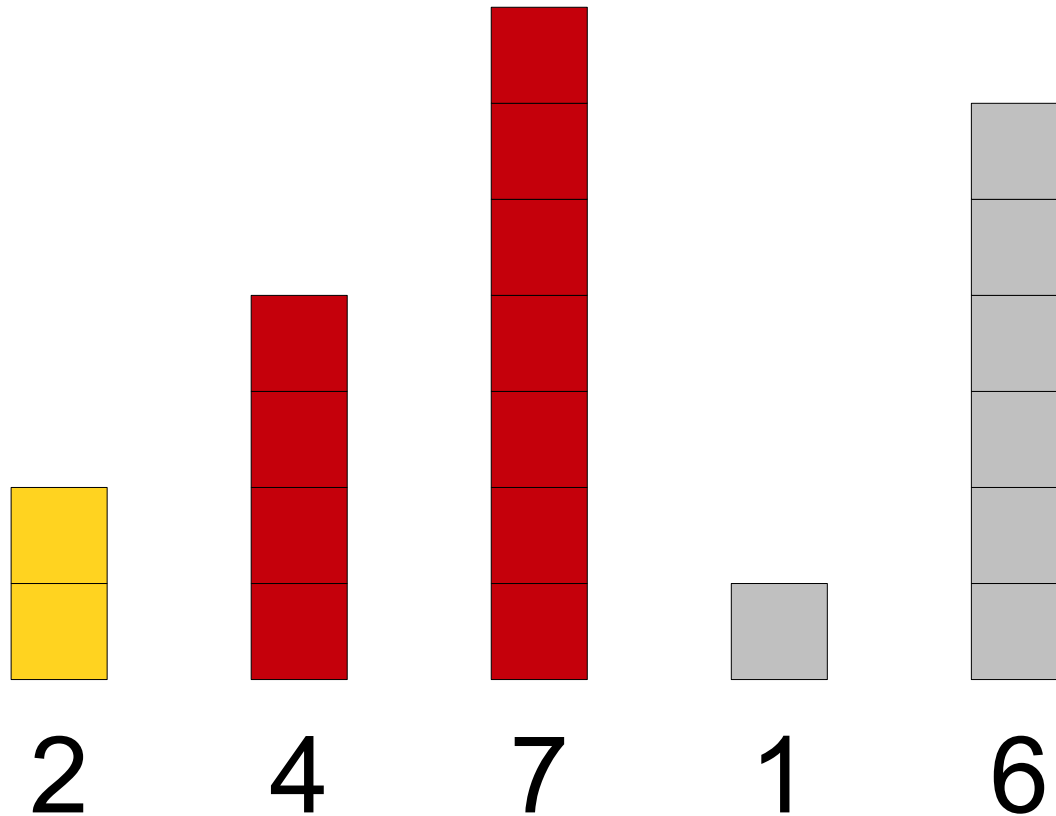
# An Initial Idea: *Insertion Sort*



**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

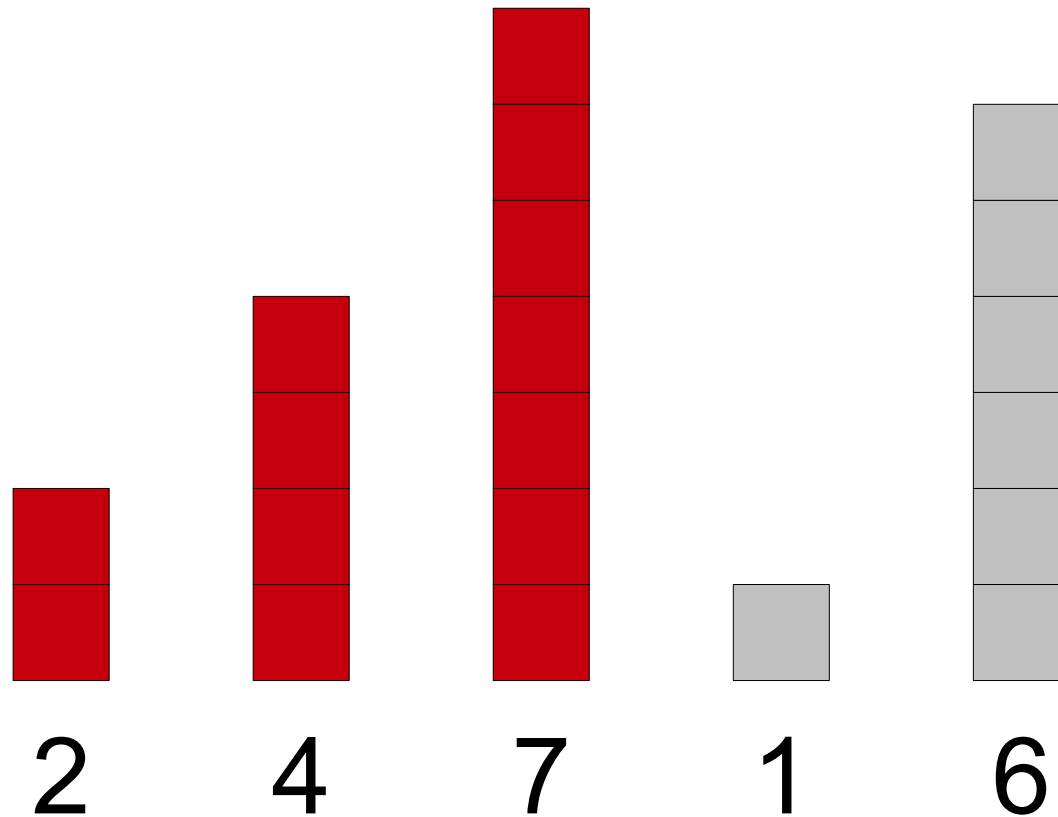


# An Initial Idea: *Insertion Sort*



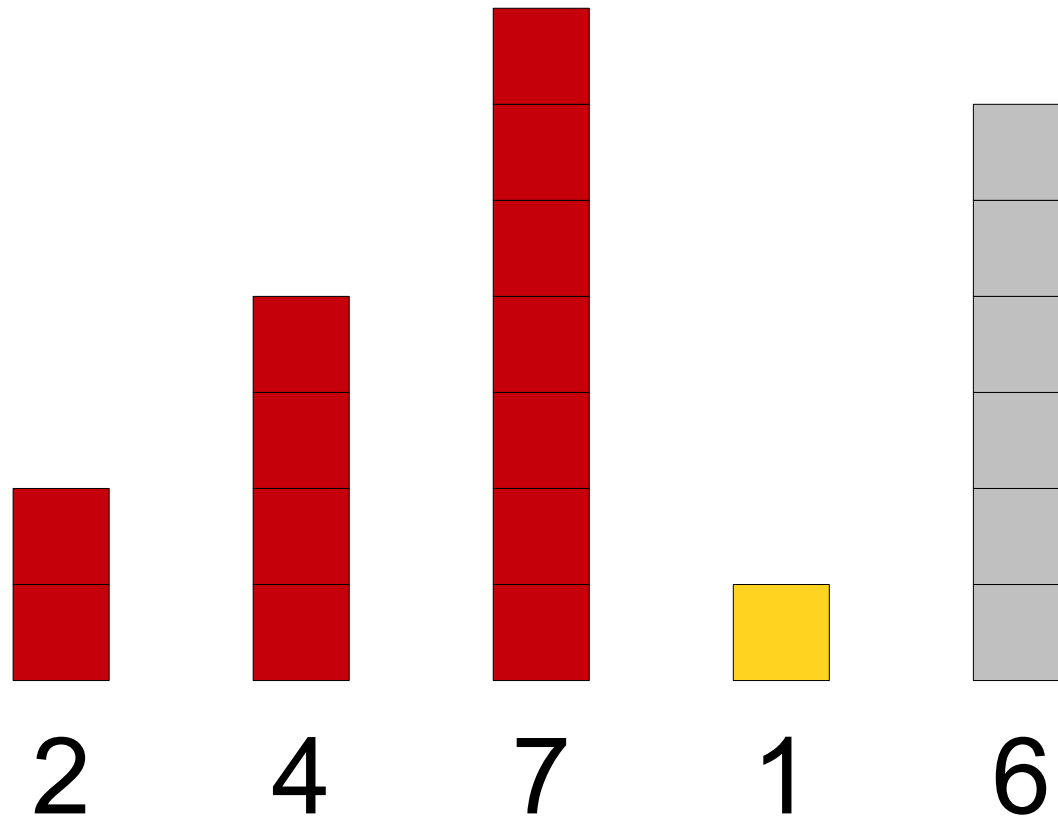
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



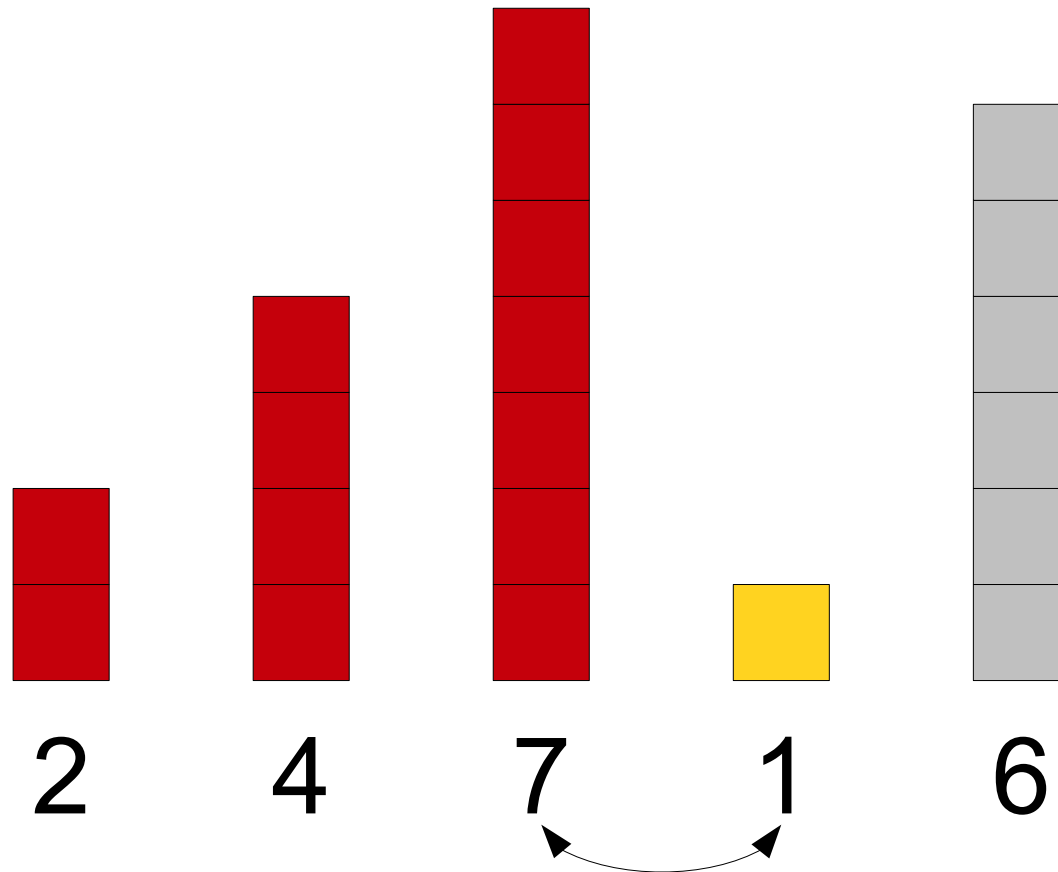
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



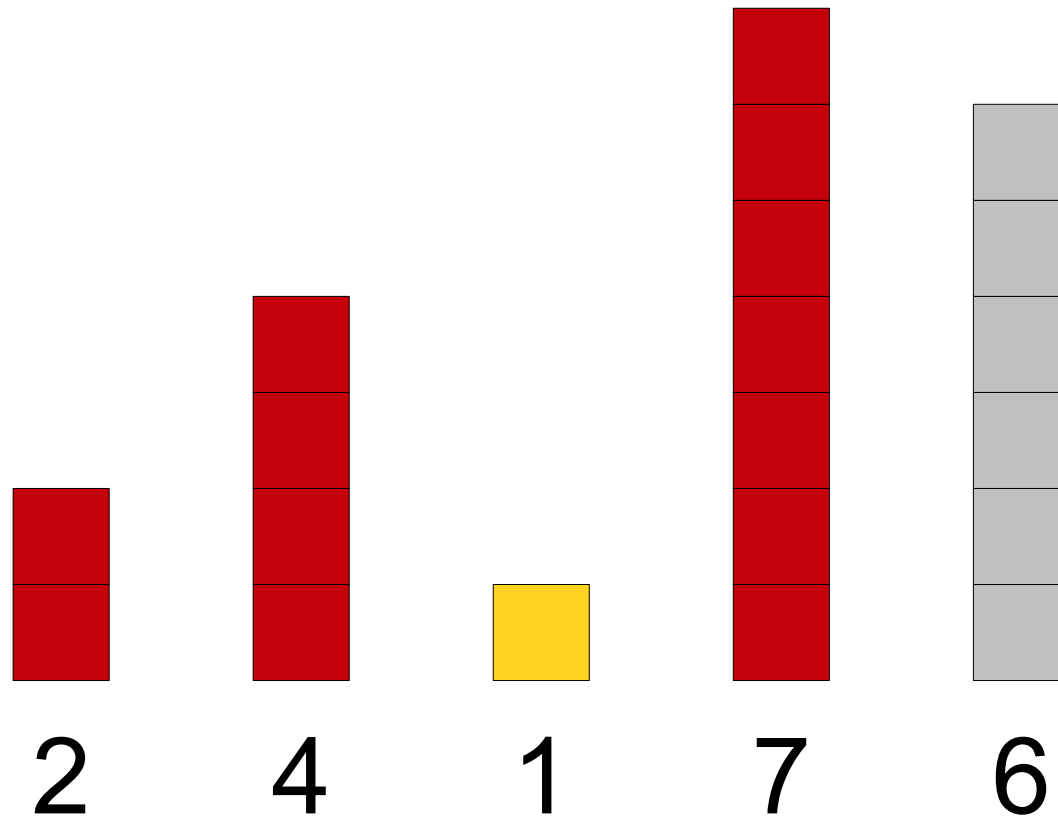
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



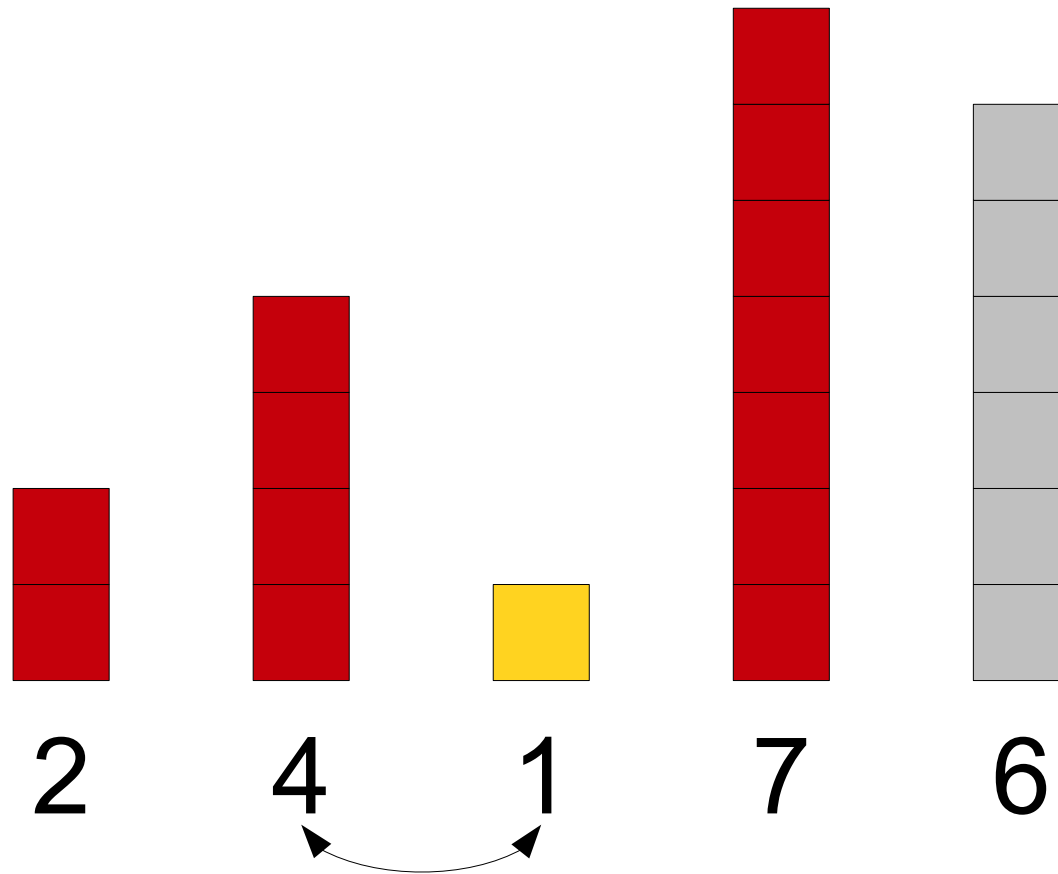
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



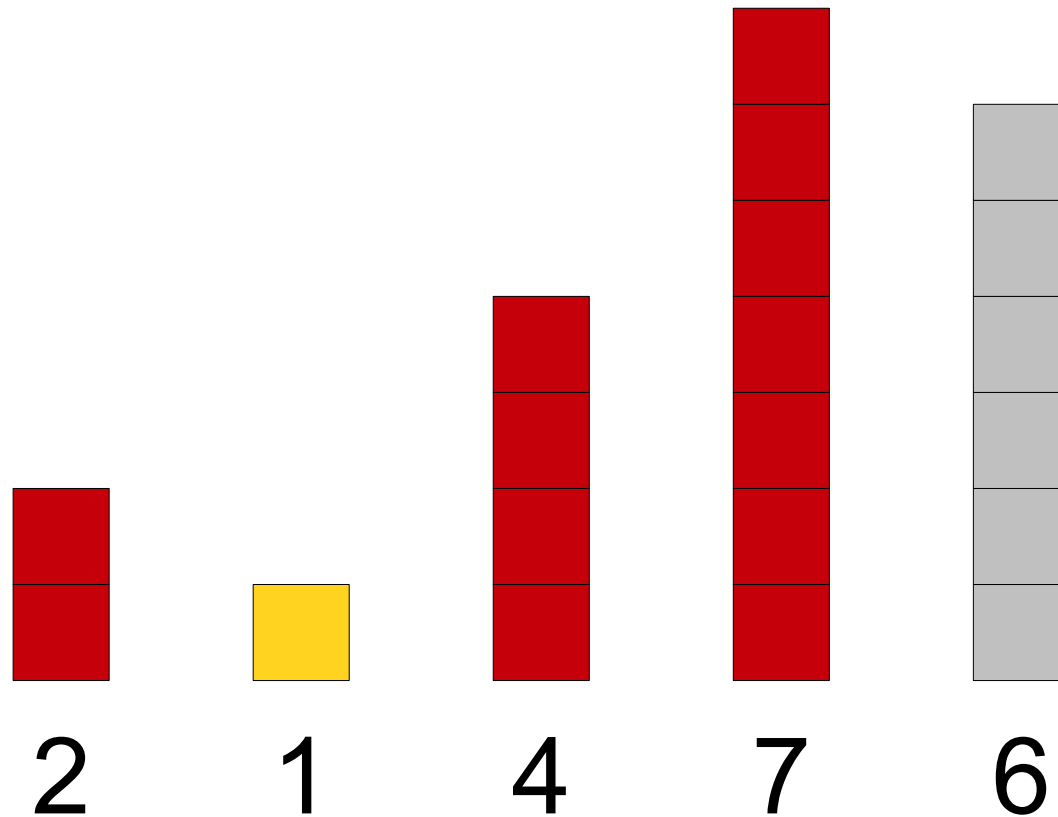
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



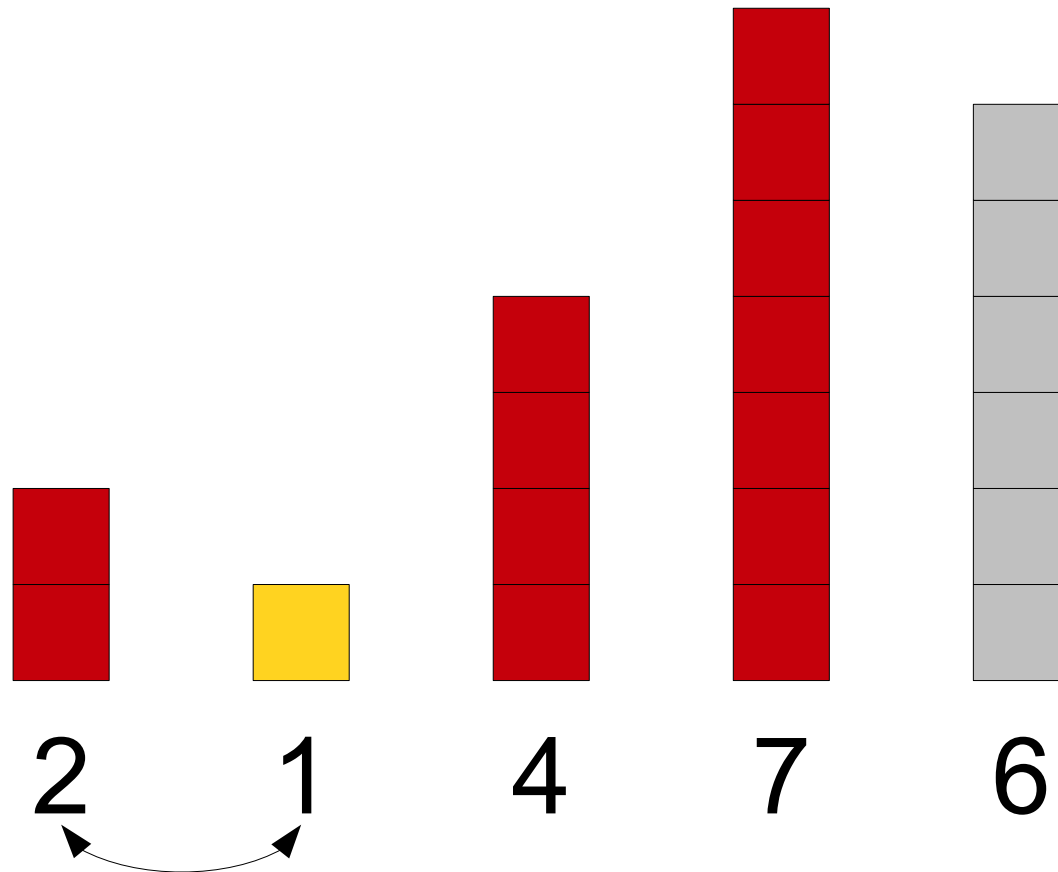
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

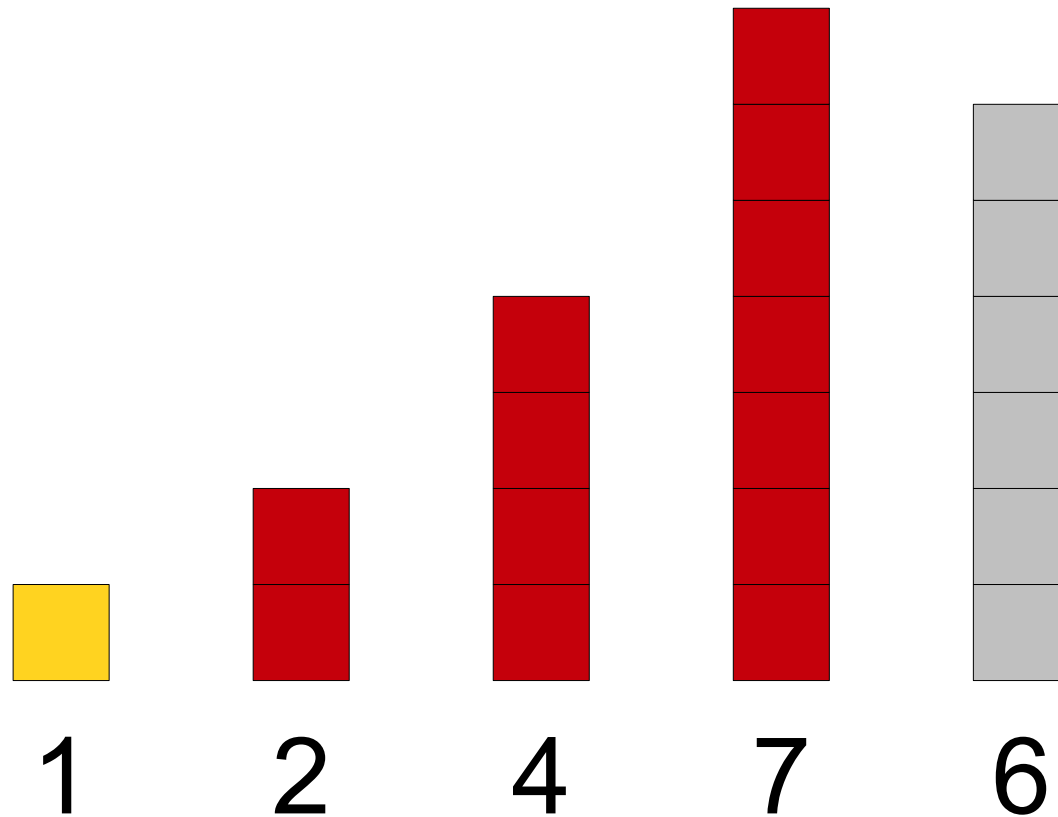
# An Initial Idea: *Insertion Sort*



**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

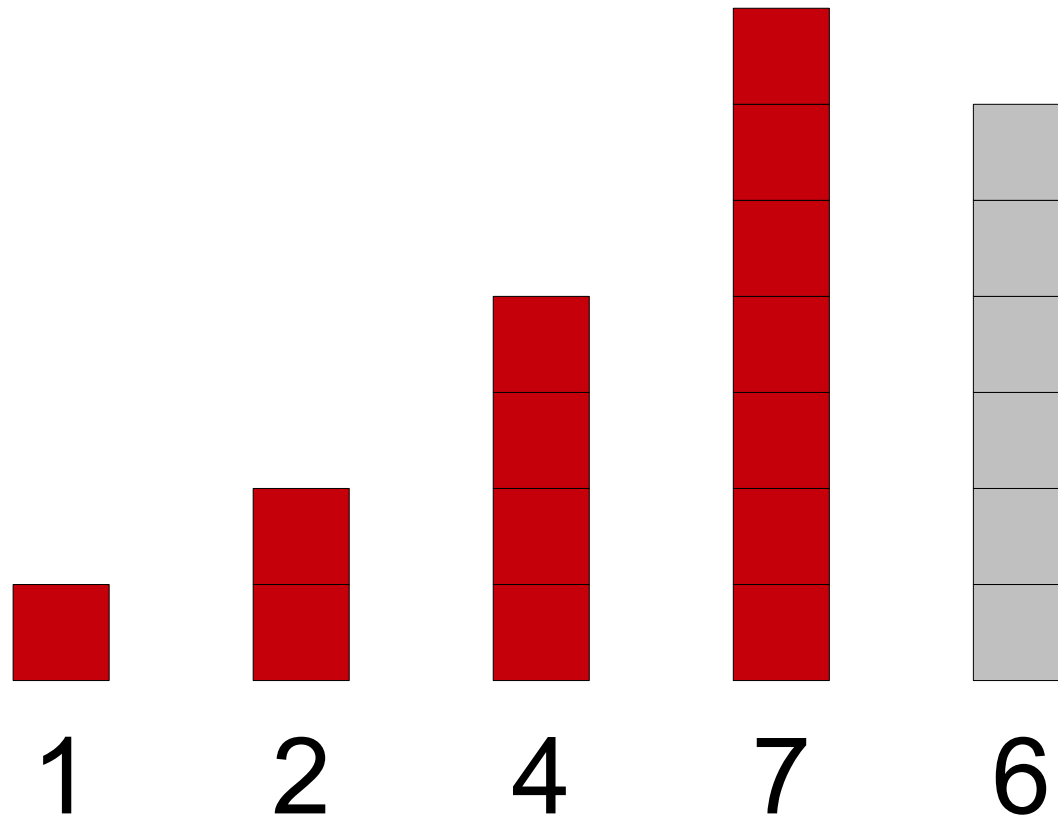


# An Initial Idea: *Insertion Sort*



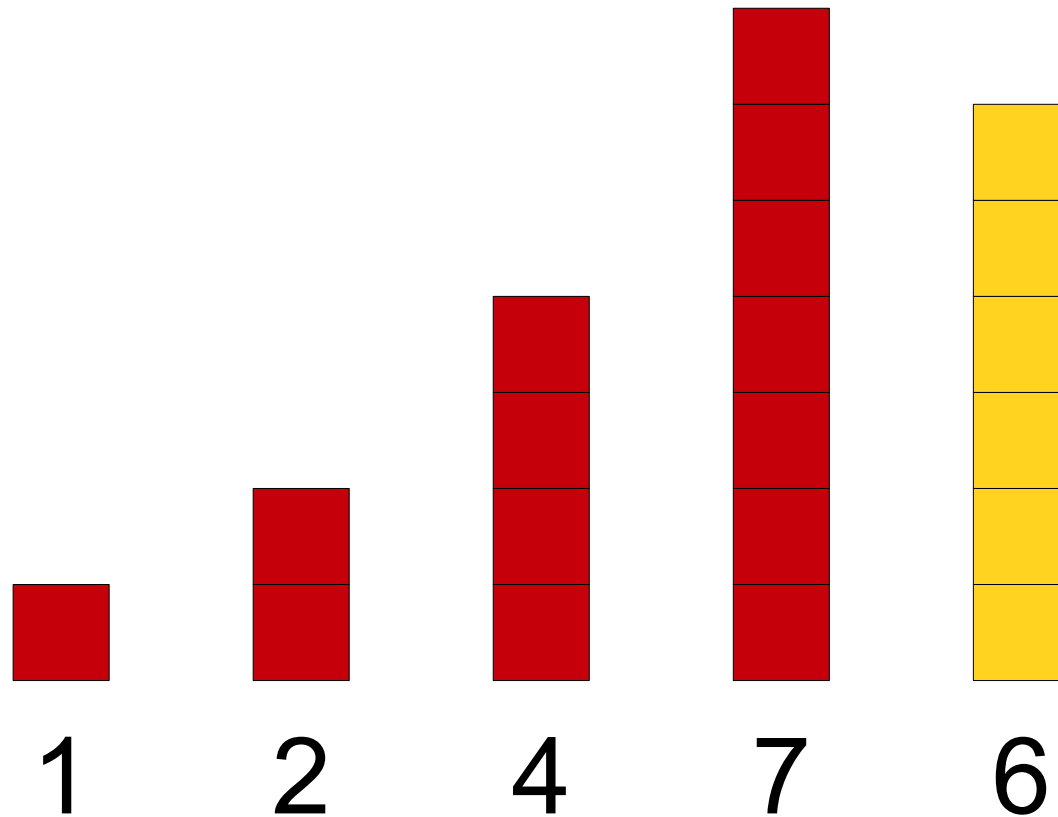
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



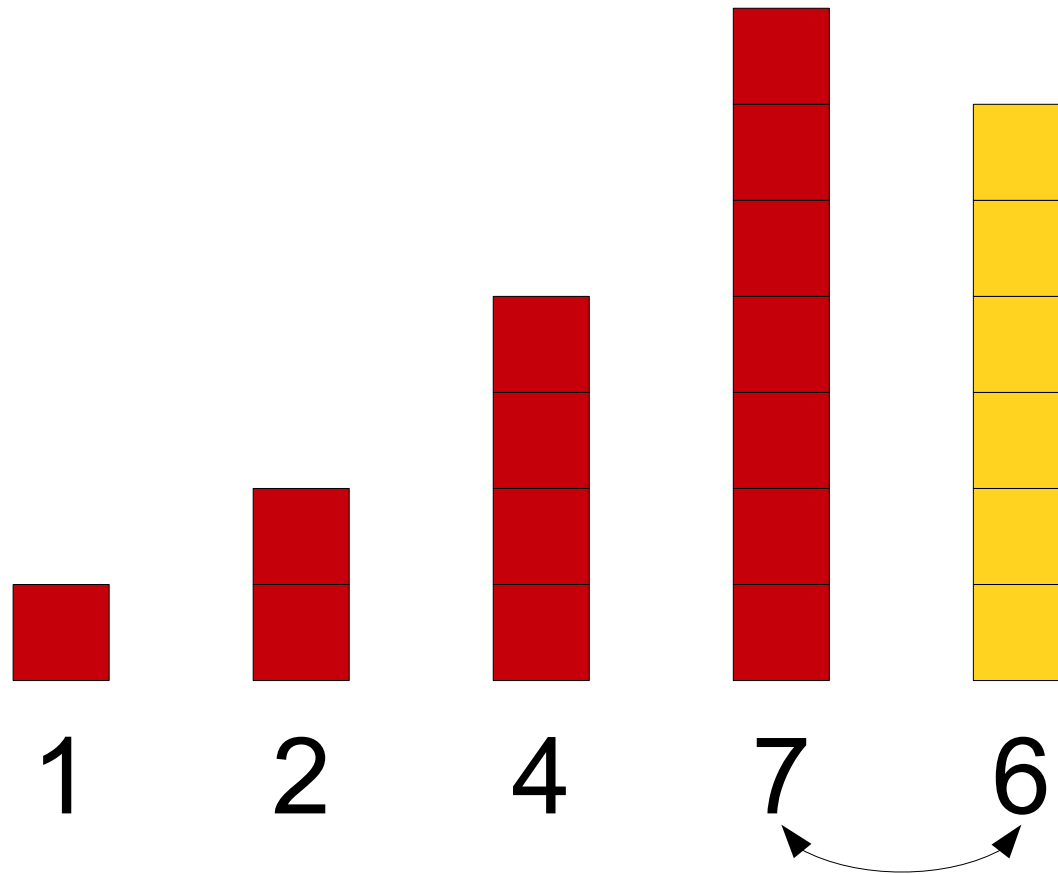
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



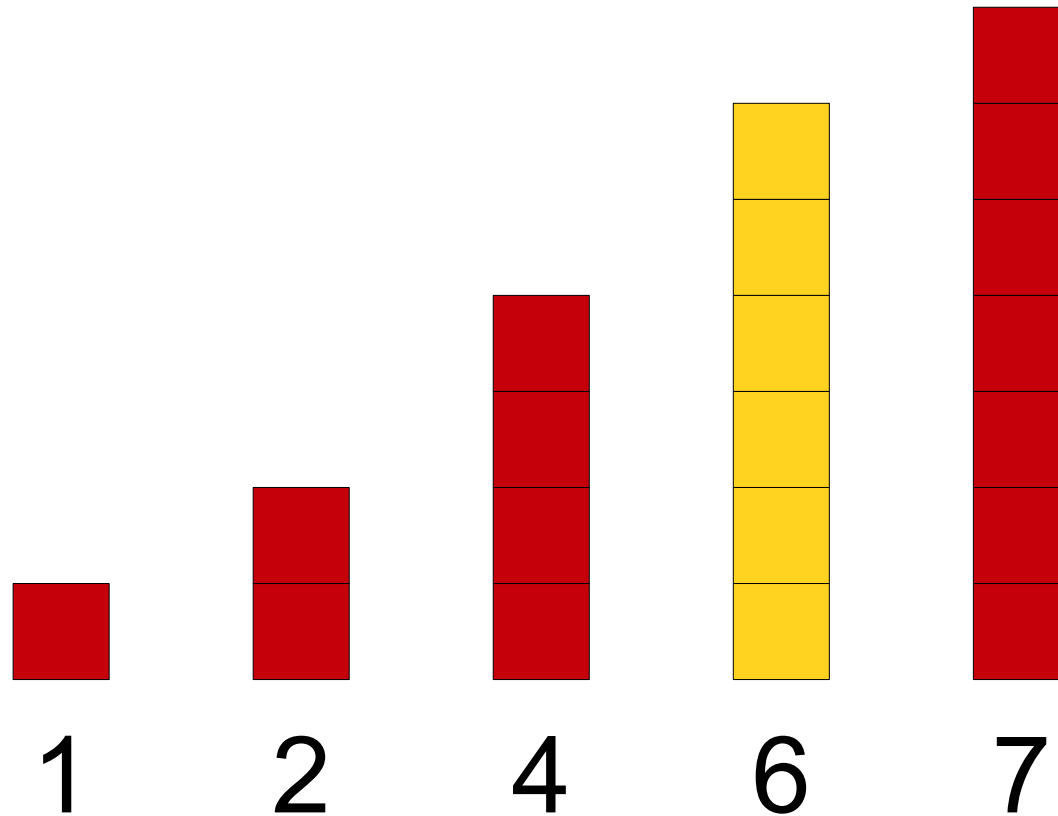
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



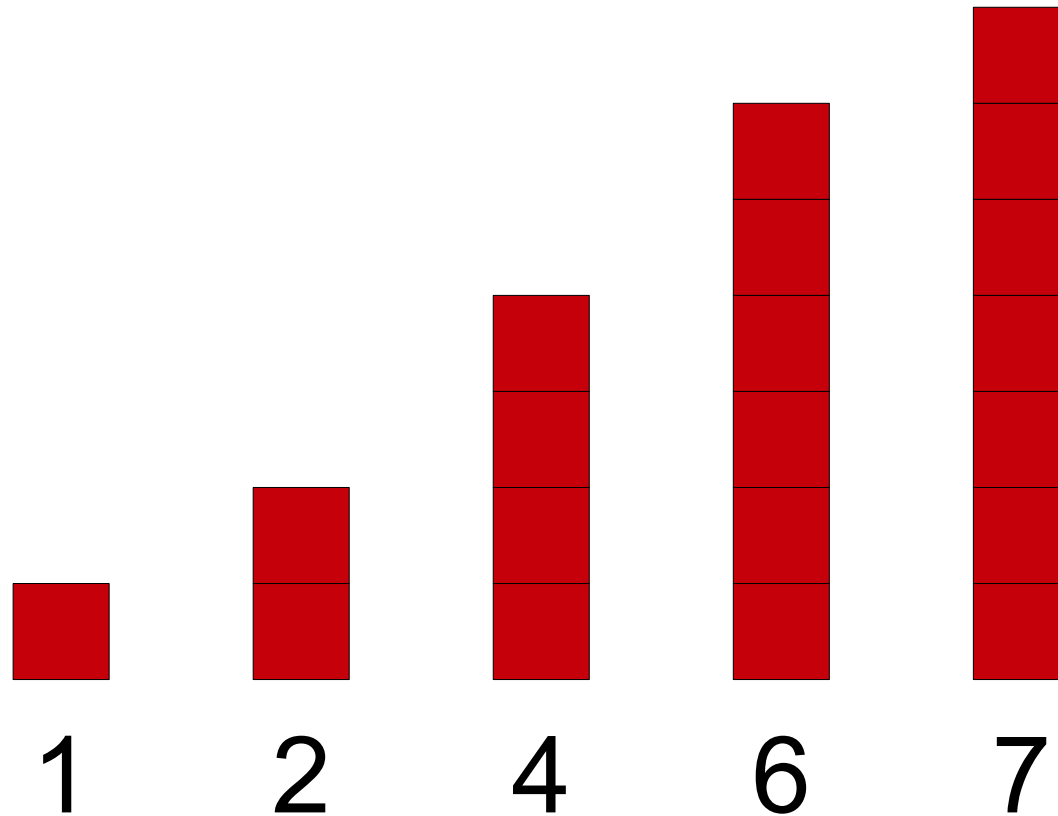
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*



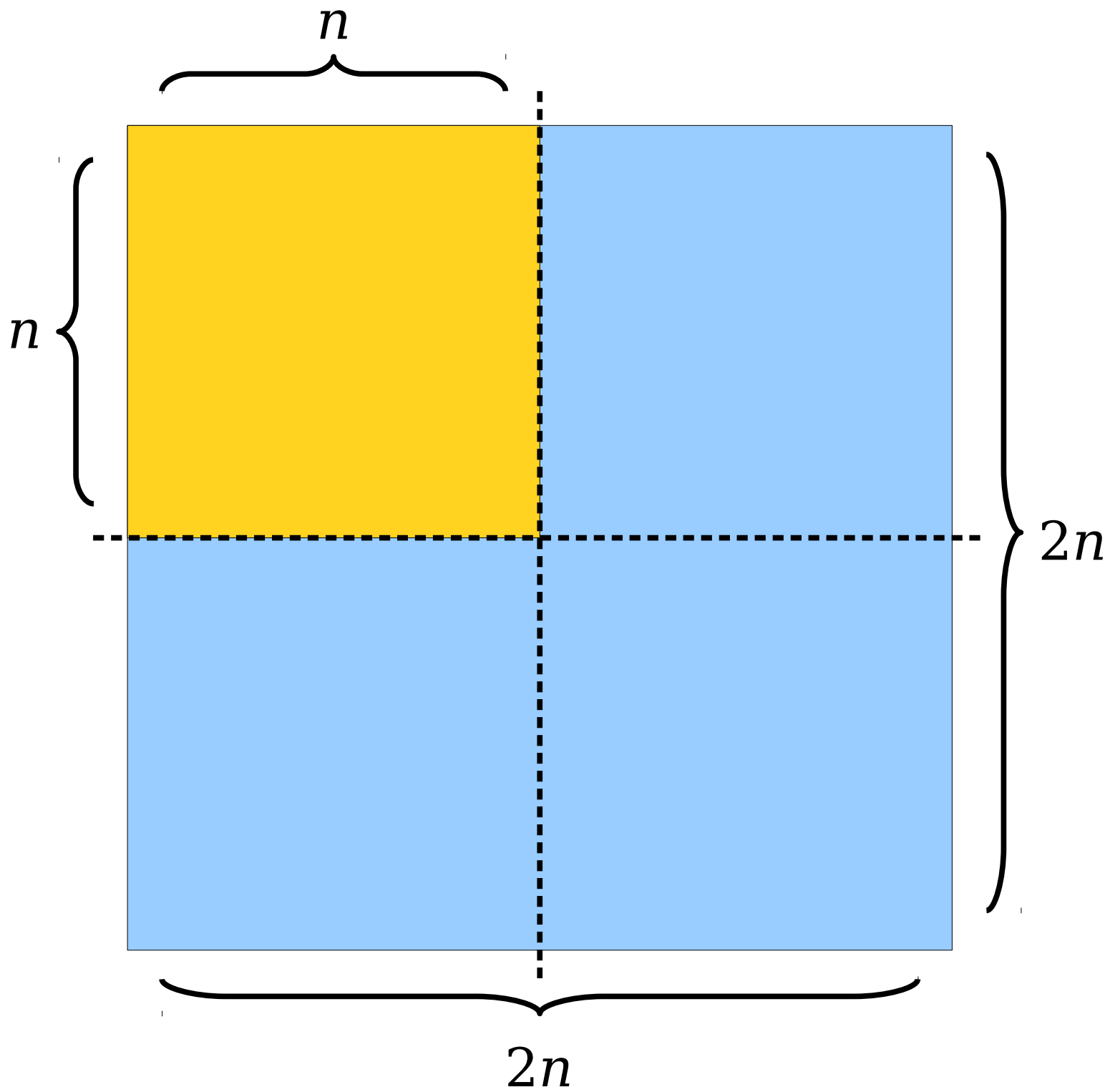
**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

# An Initial Idea: *Insertion Sort*

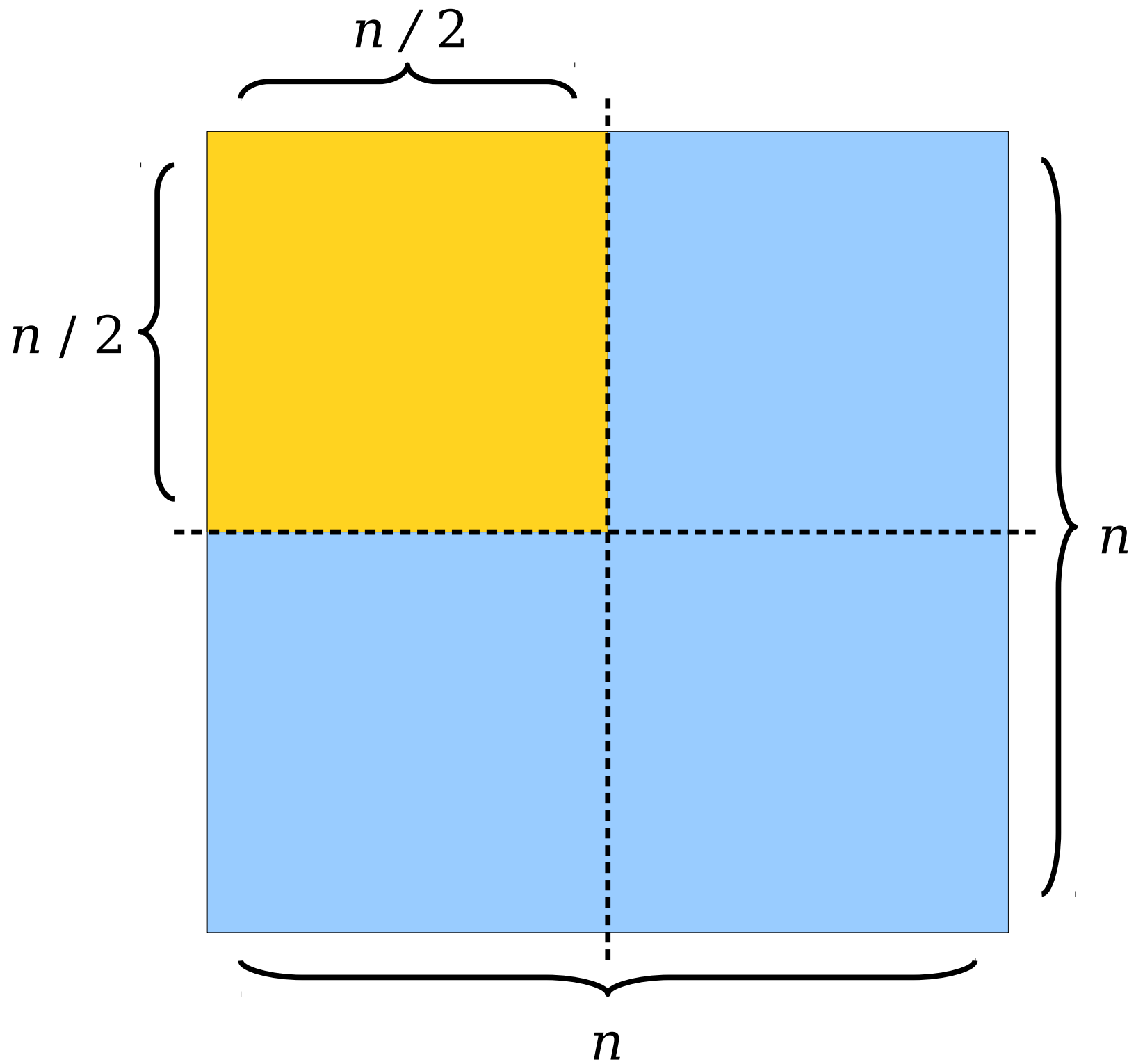


**Rule:** Swap each element to the left until it doesn't have a bigger element before it.

New Stuff!







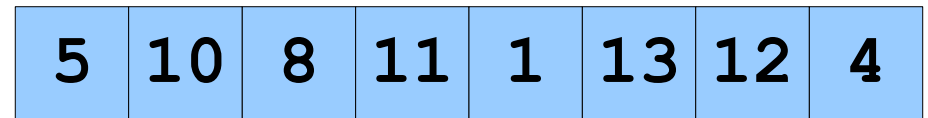
# Thinking About $O(n^2)$



$T(n)$



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



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

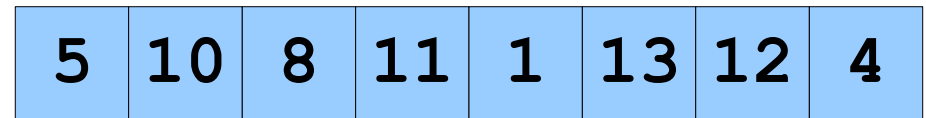
# Thinking About $O(n^2)$



$T(n)$



$\frac{1}{4}T(n)$



$\frac{1}{4}T(n)$

# Thinking About $O(n^2)$

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

$T(n)$

2	3	6	7	9	14	15	16
---	---	---	---	---	----	----	----

$\frac{1}{4}T(n)$

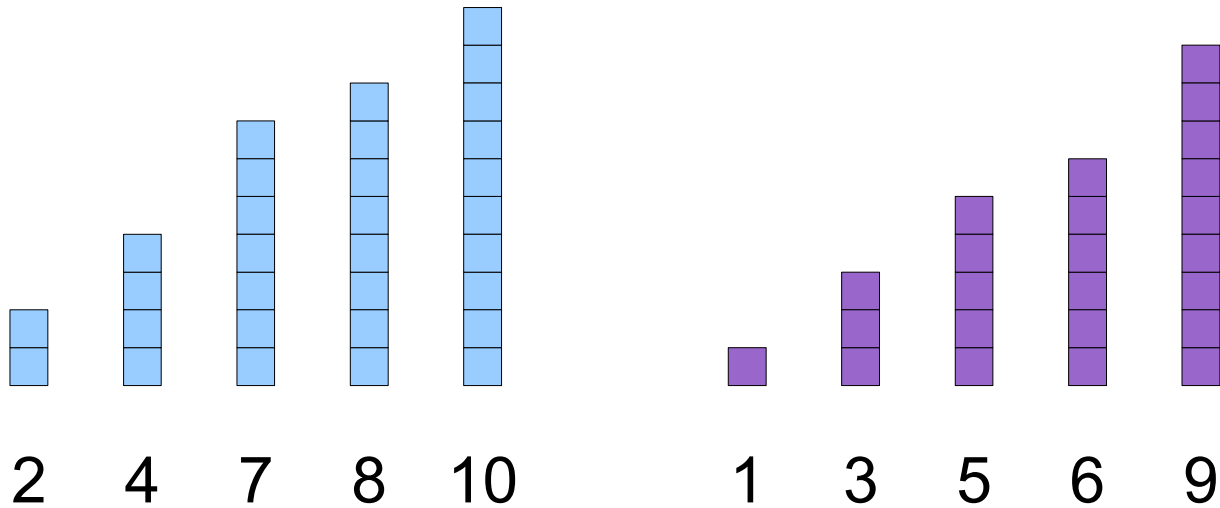
1	4	5	8	10	11	12	13
---	---	---	---	----	----	----	----

$\frac{1}{4}T(n)$

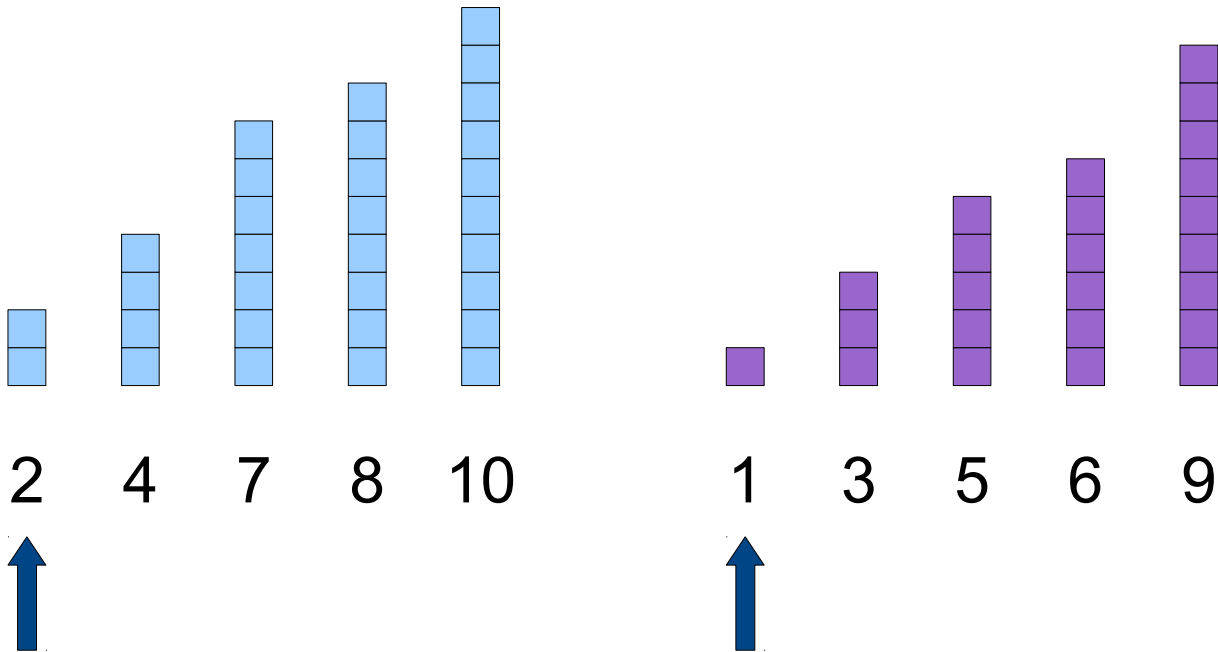
$$2 \cdot \frac{1}{4}T(n) = \frac{1}{2}T(n)$$

The Key Insight: ***Merge***

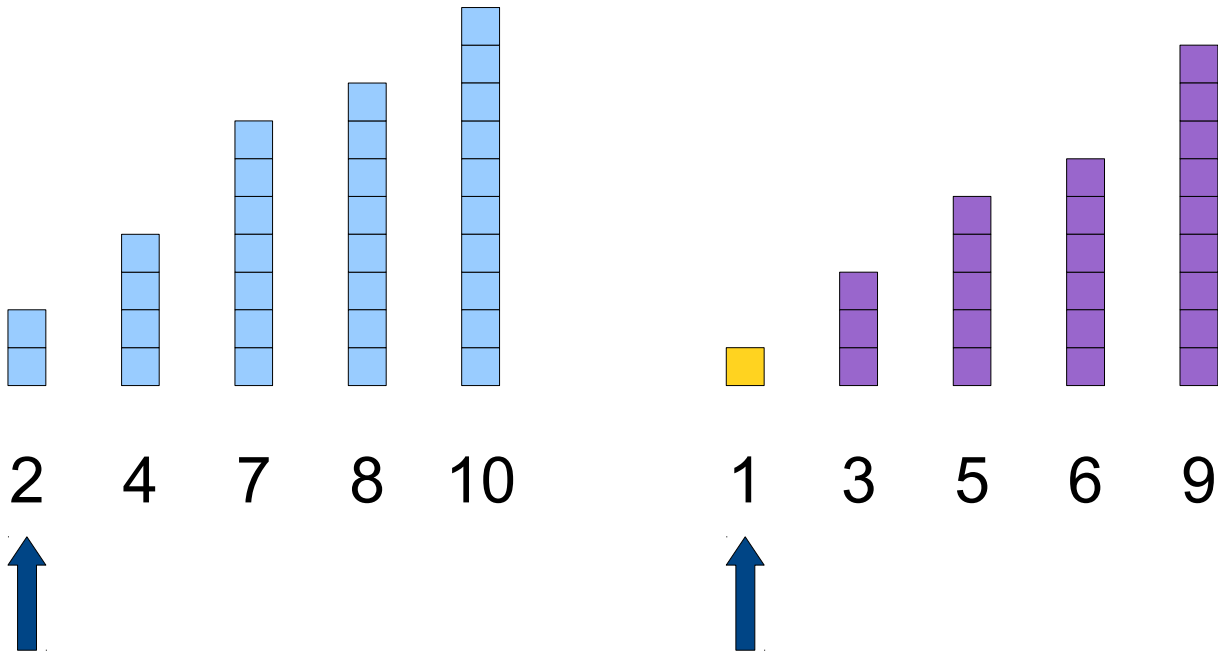
# The Key Insight: *Merge*



# The Key Insight: *Merge*

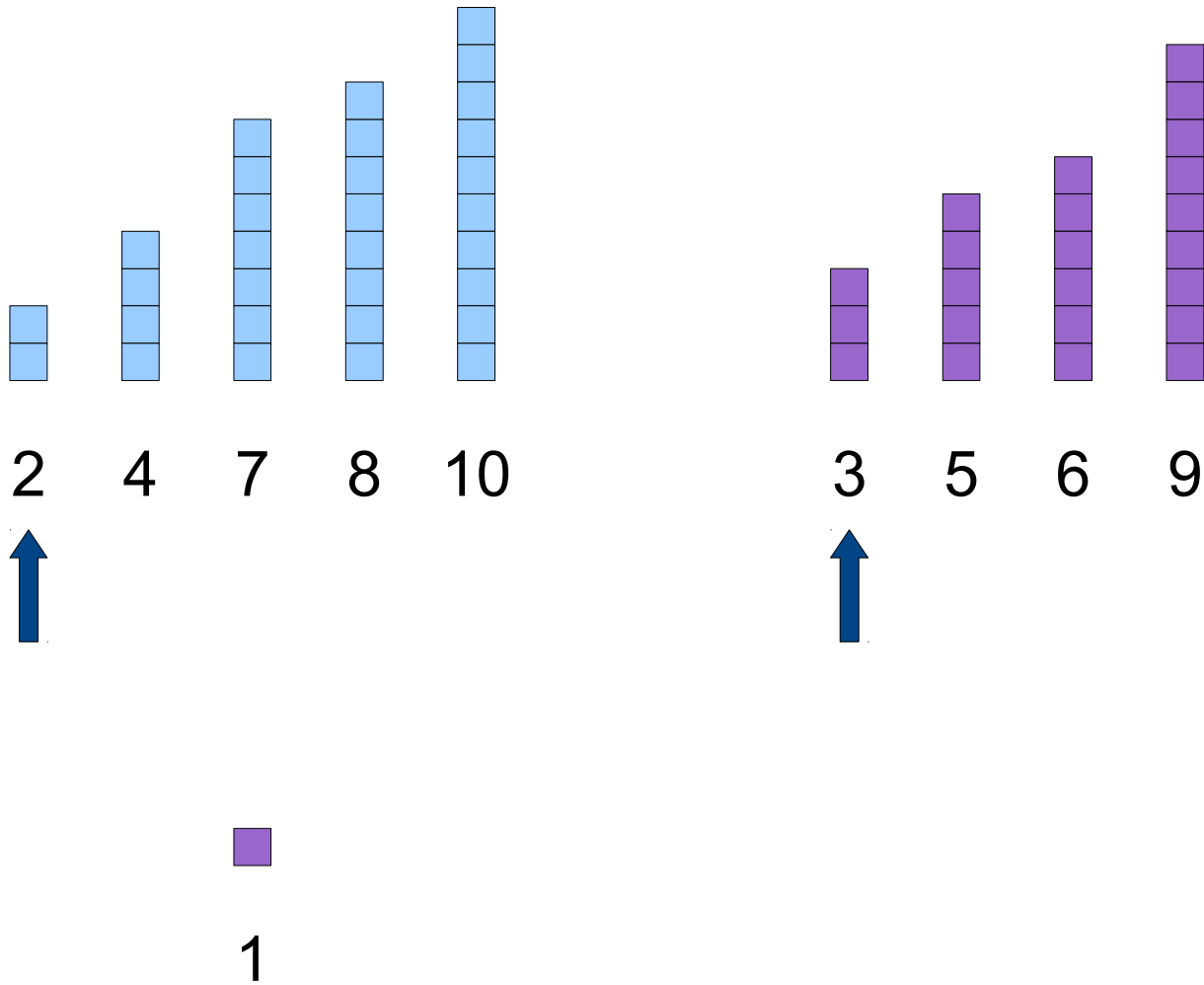


# The Key Insight: *Merge*

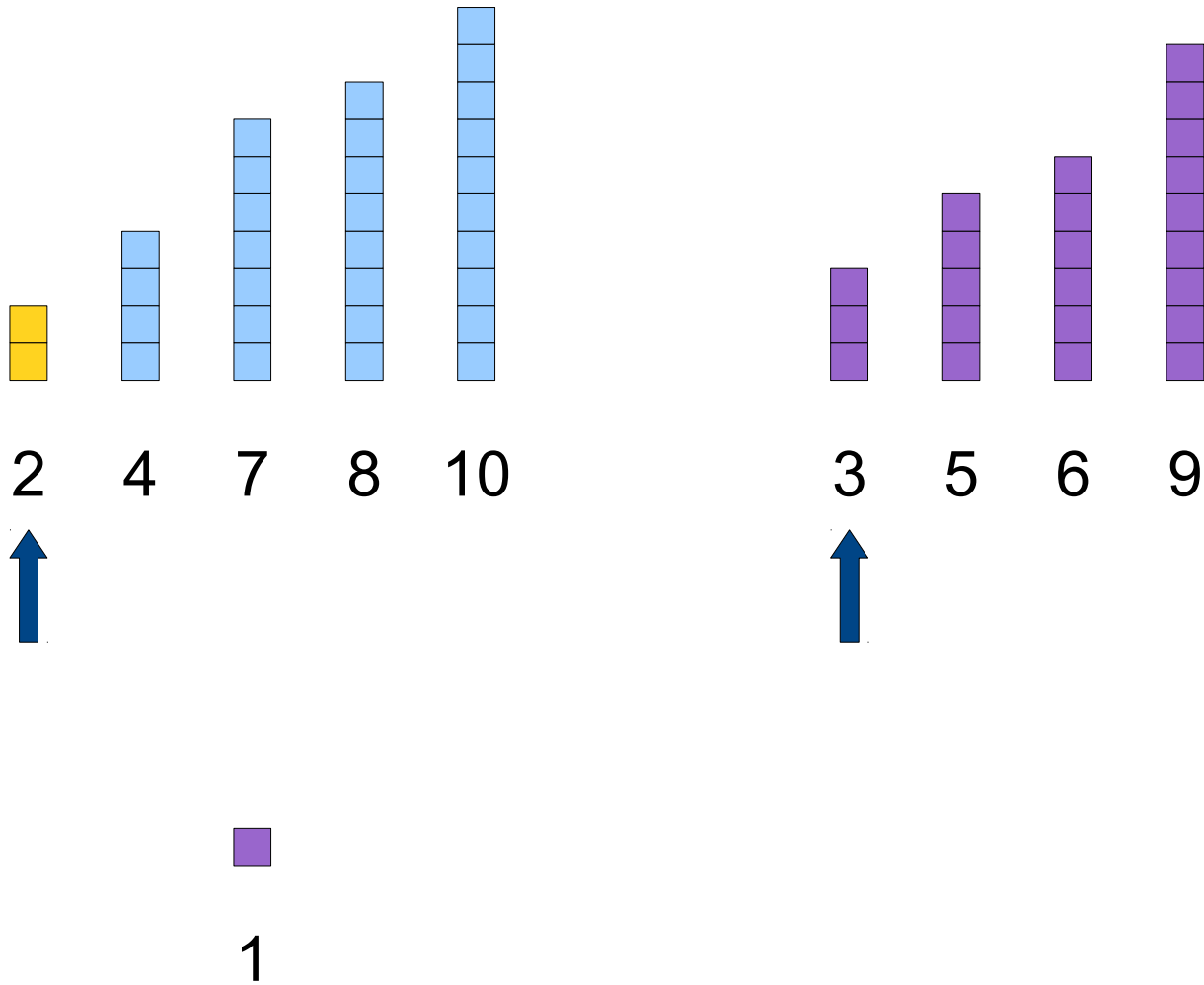




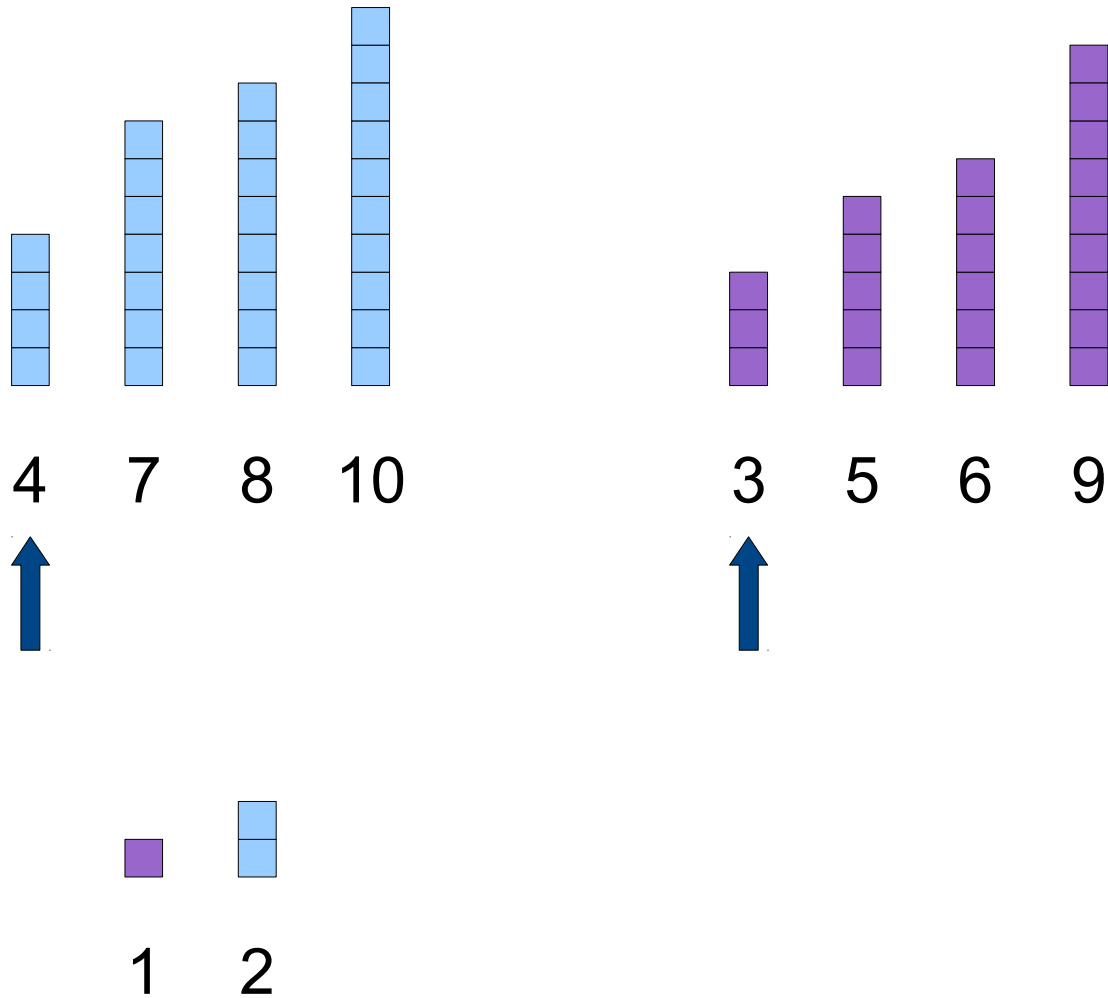
# The Key Insight: *Merge*



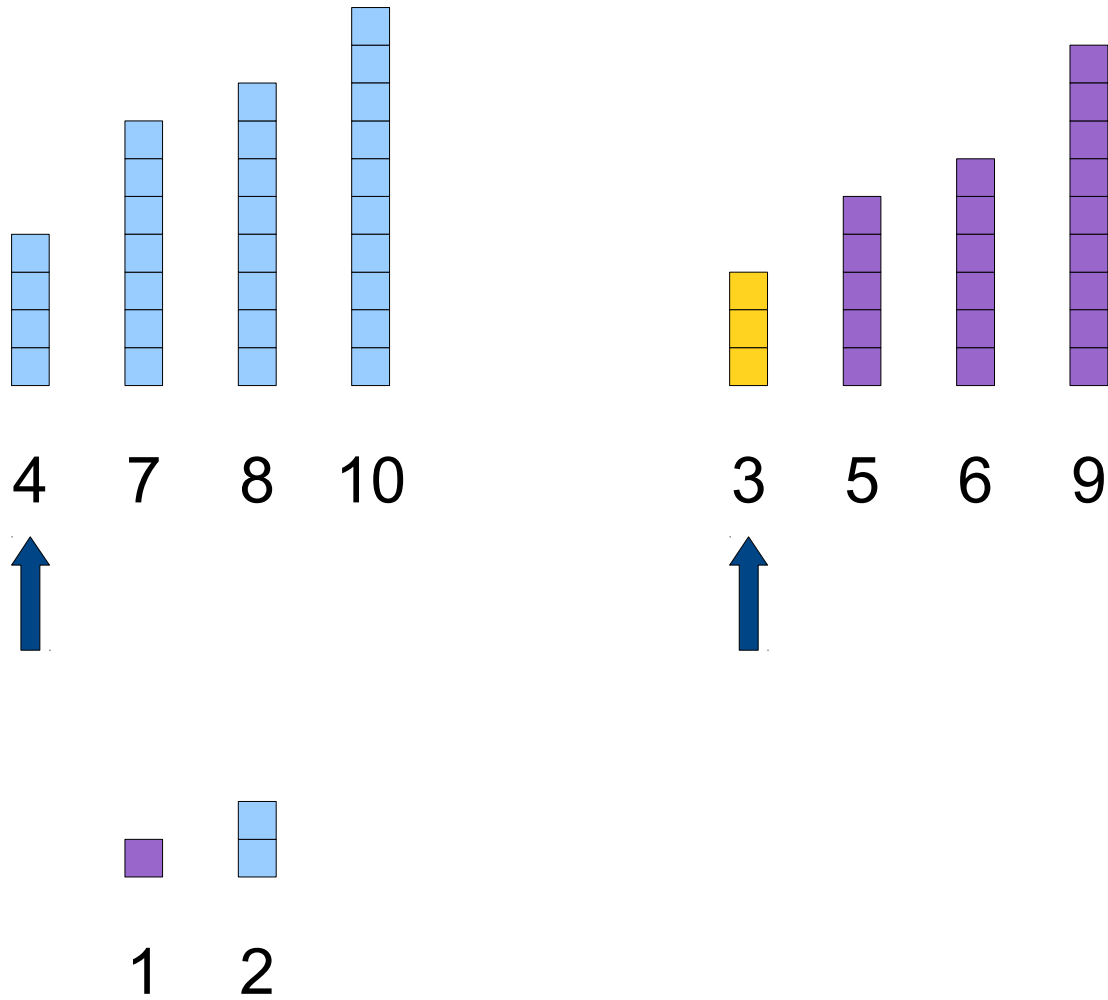
# The Key Insight: *Merge*



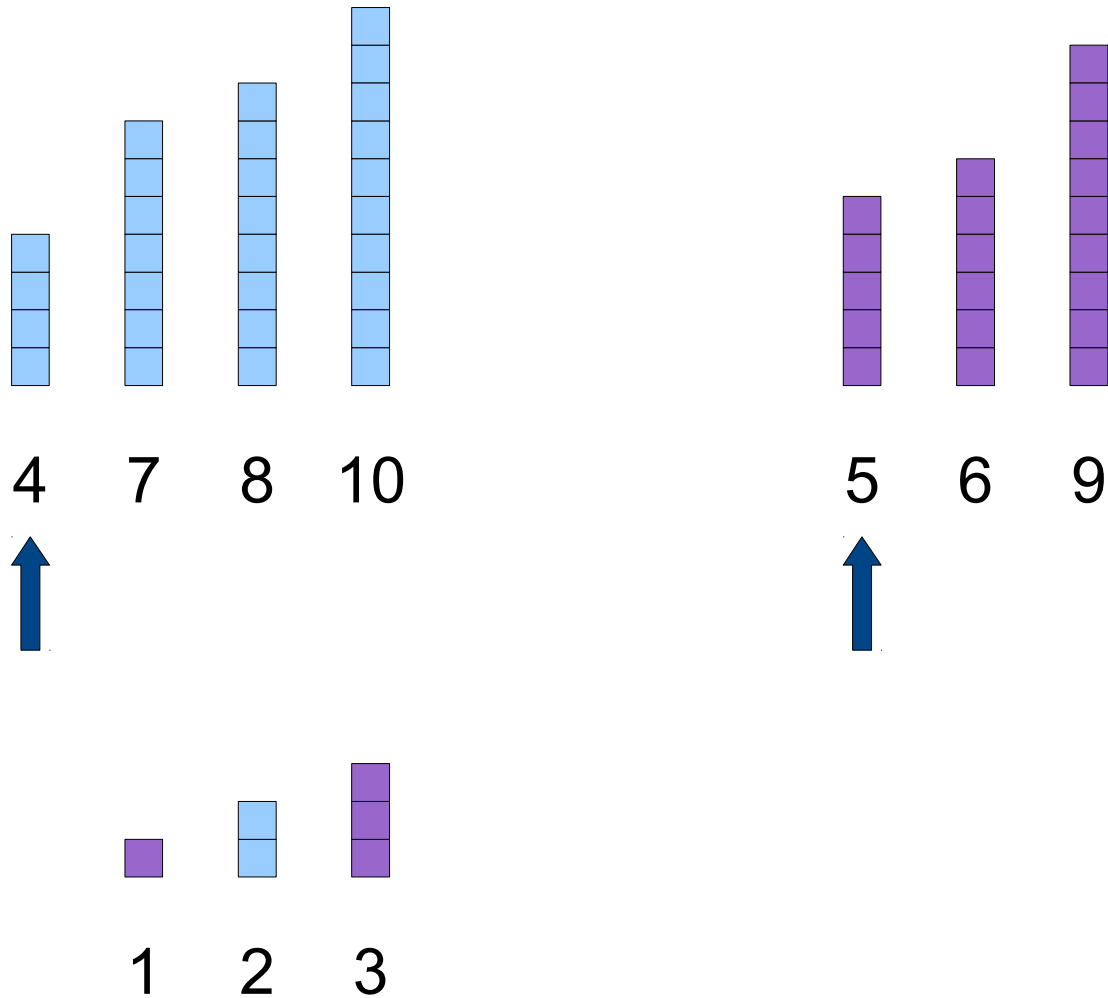
# The Key Insight: *Merge*



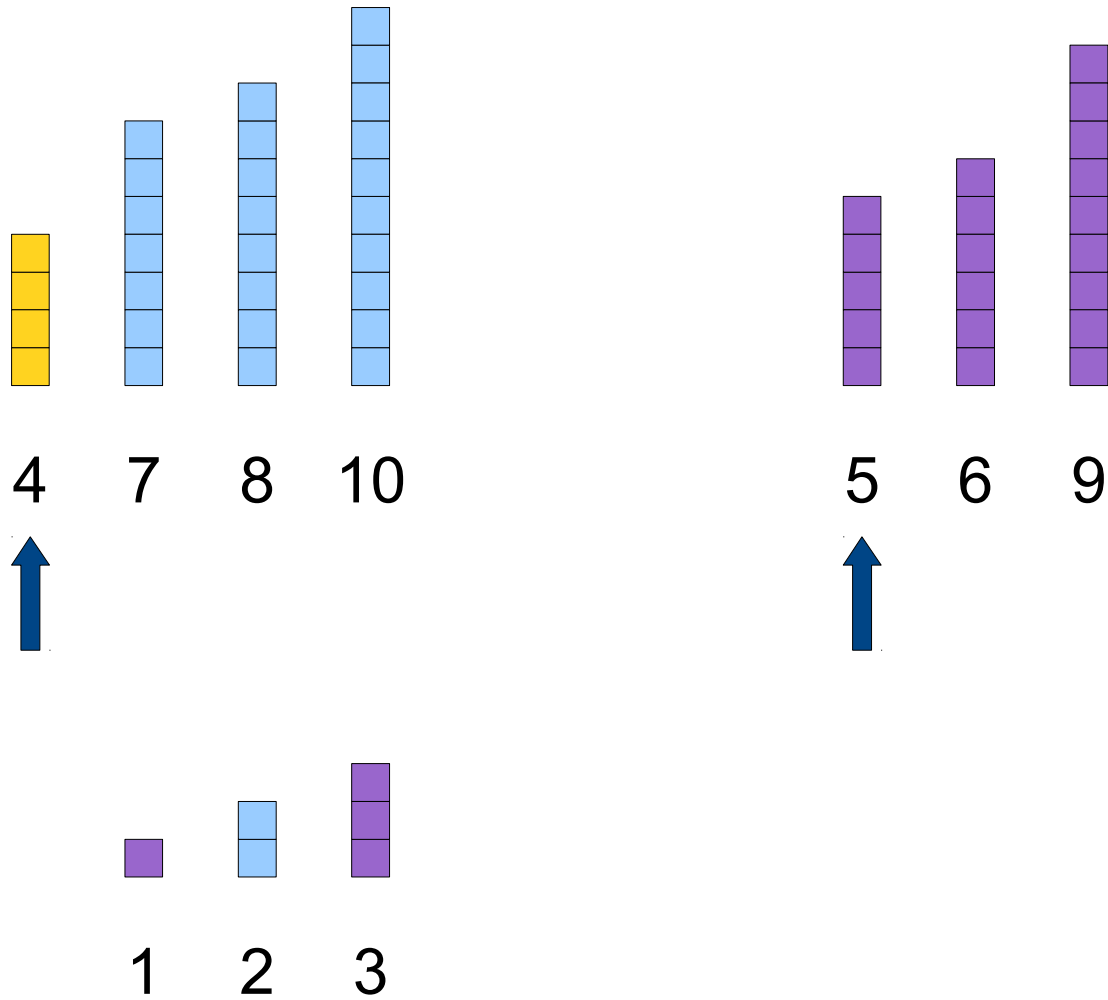
# The Key Insight: *Merge*



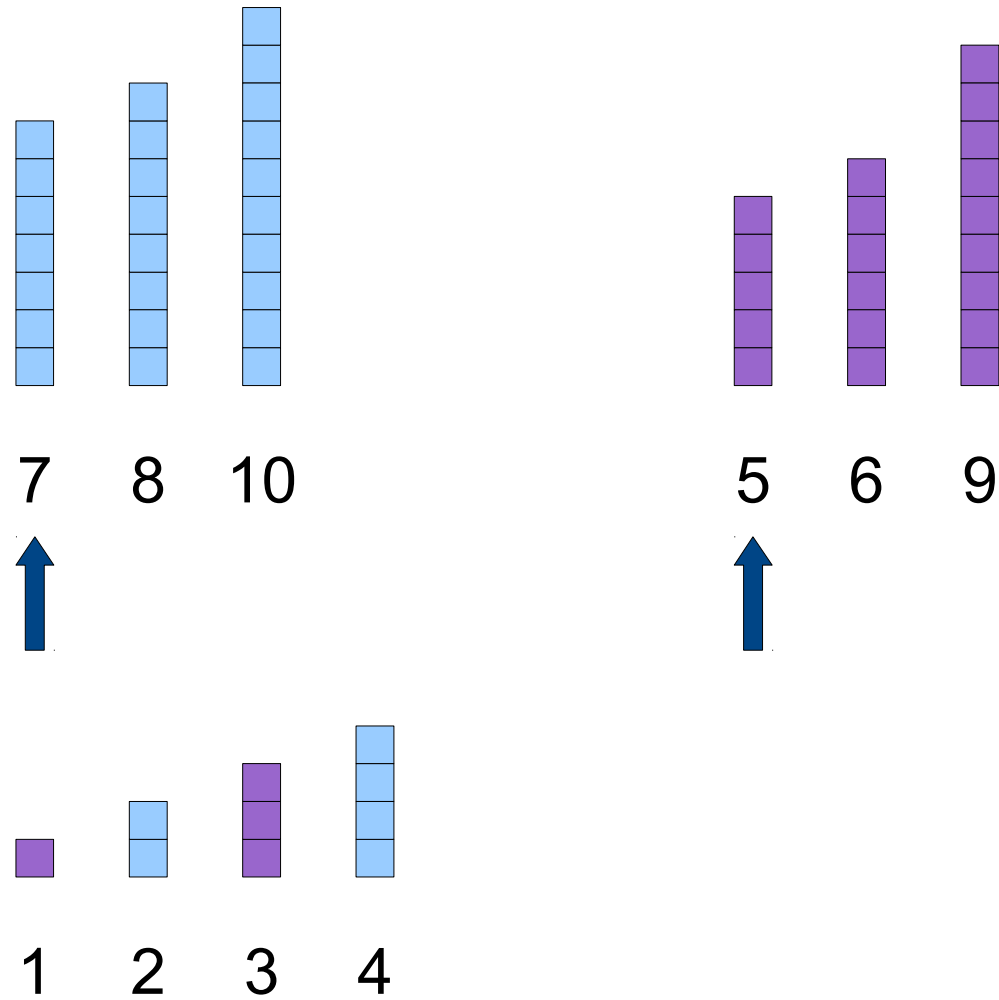
# The Key Insight: *Merge*



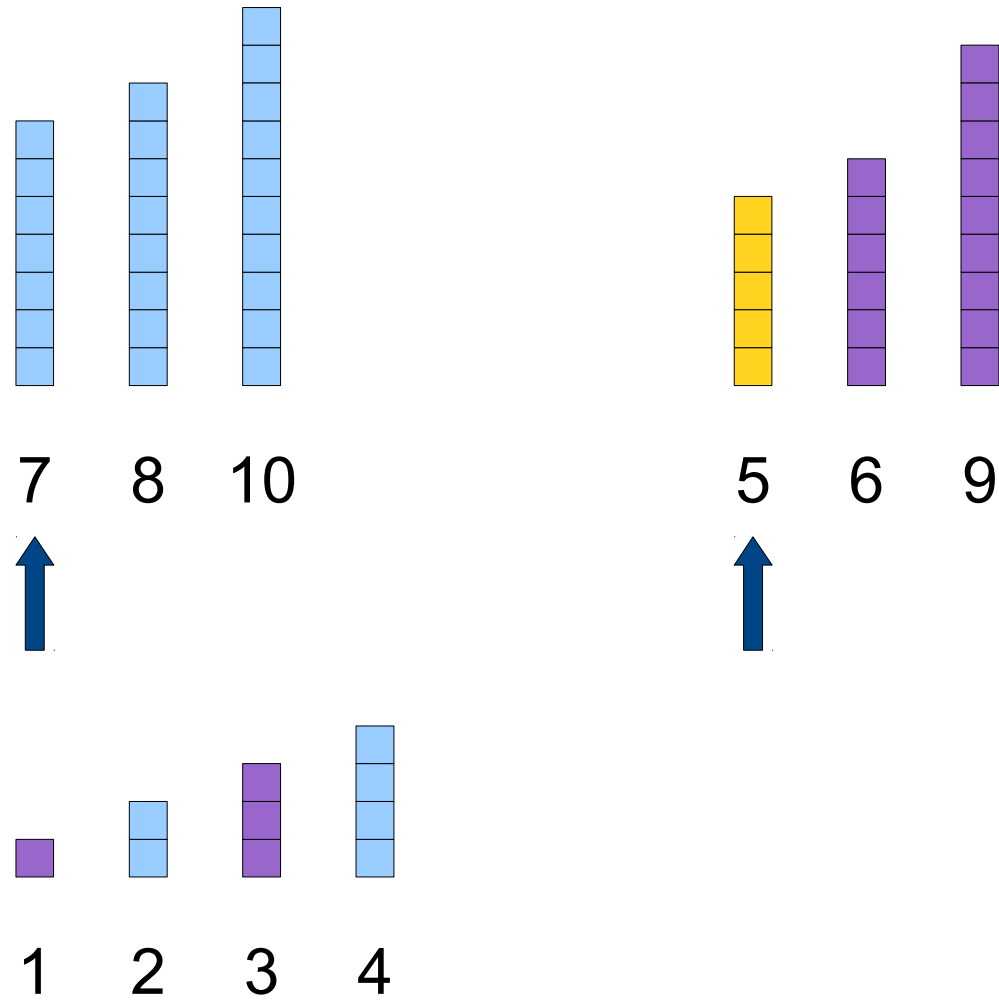
# The Key Insight: *Merge*



# The Key Insight: *Merge*

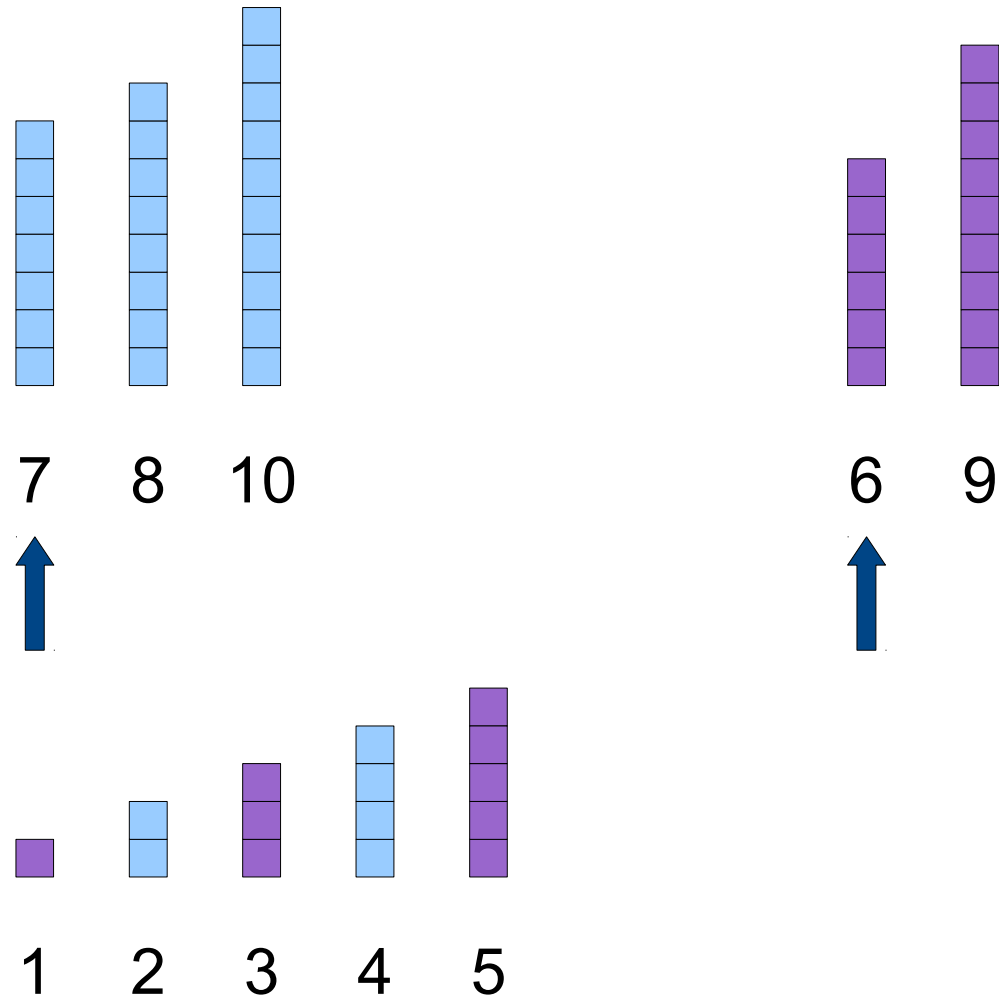


# The Key Insight: *Merge*

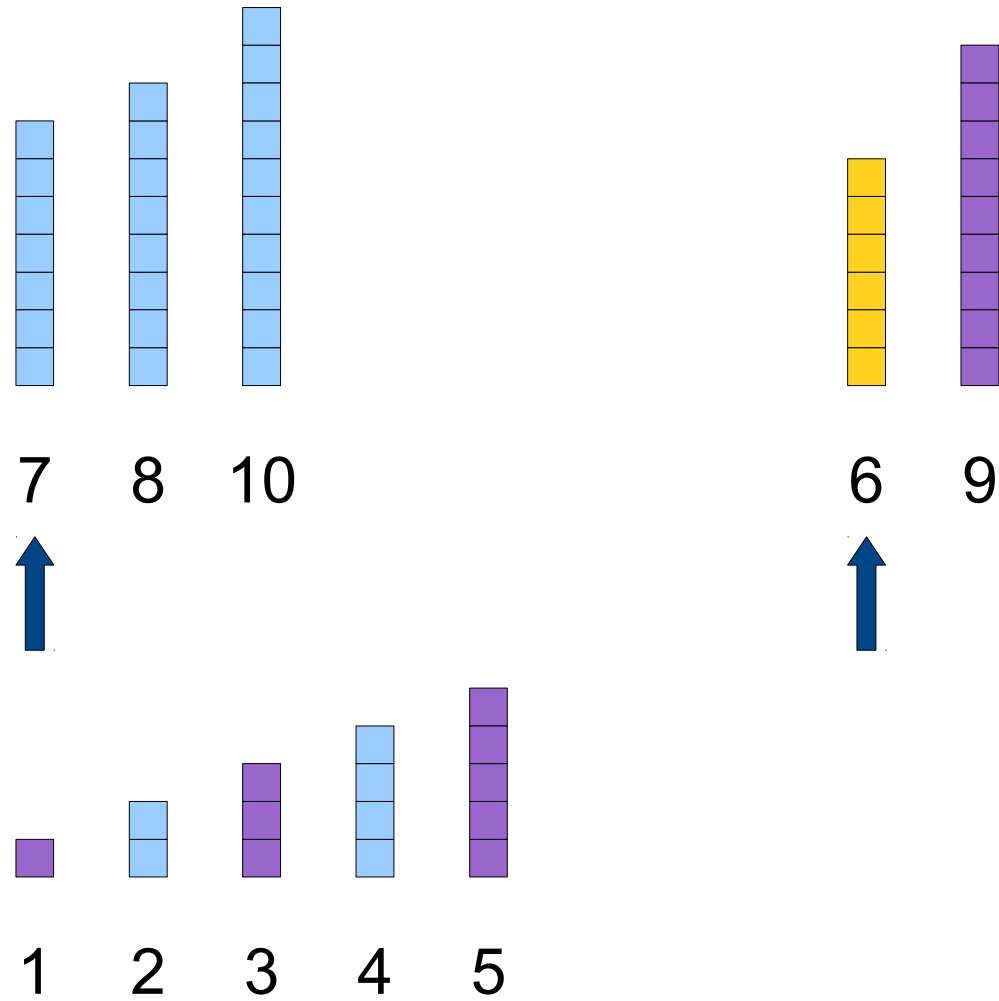




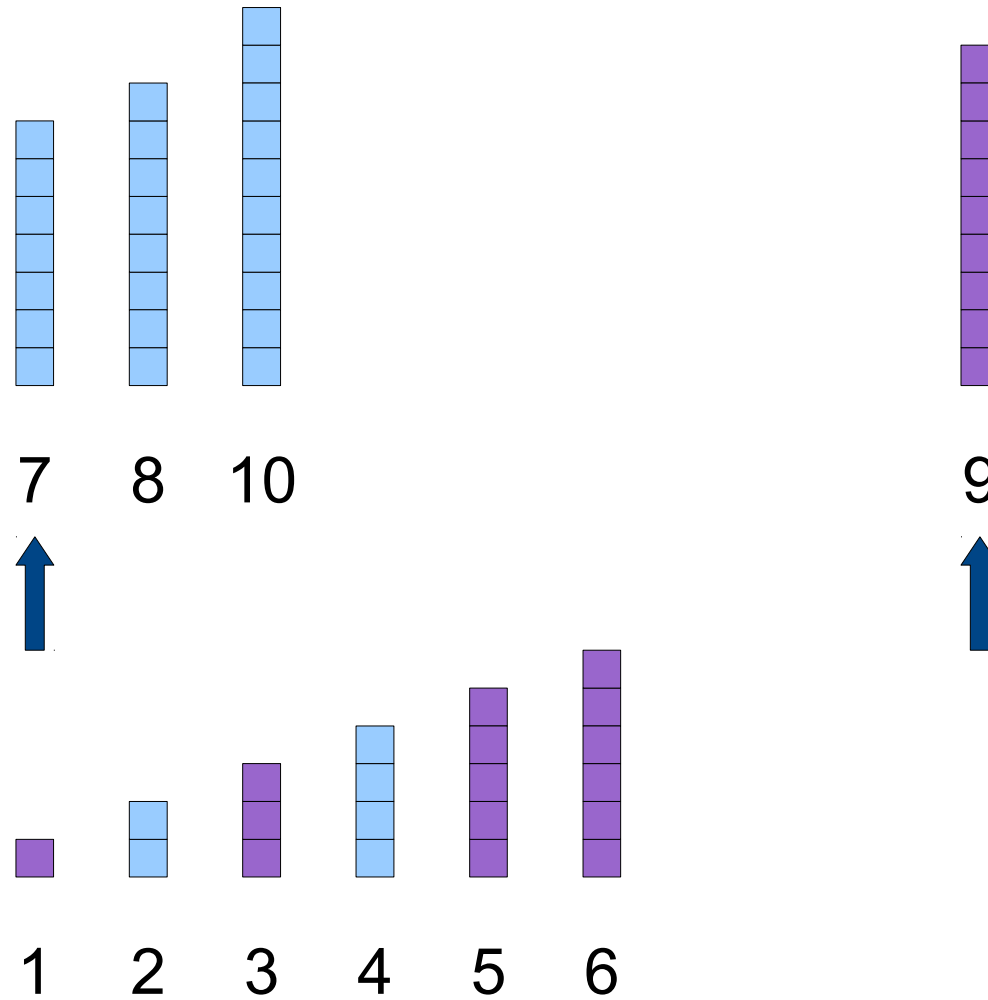
# The Key Insight: *Merge*



# The Key Insight: *Merge*

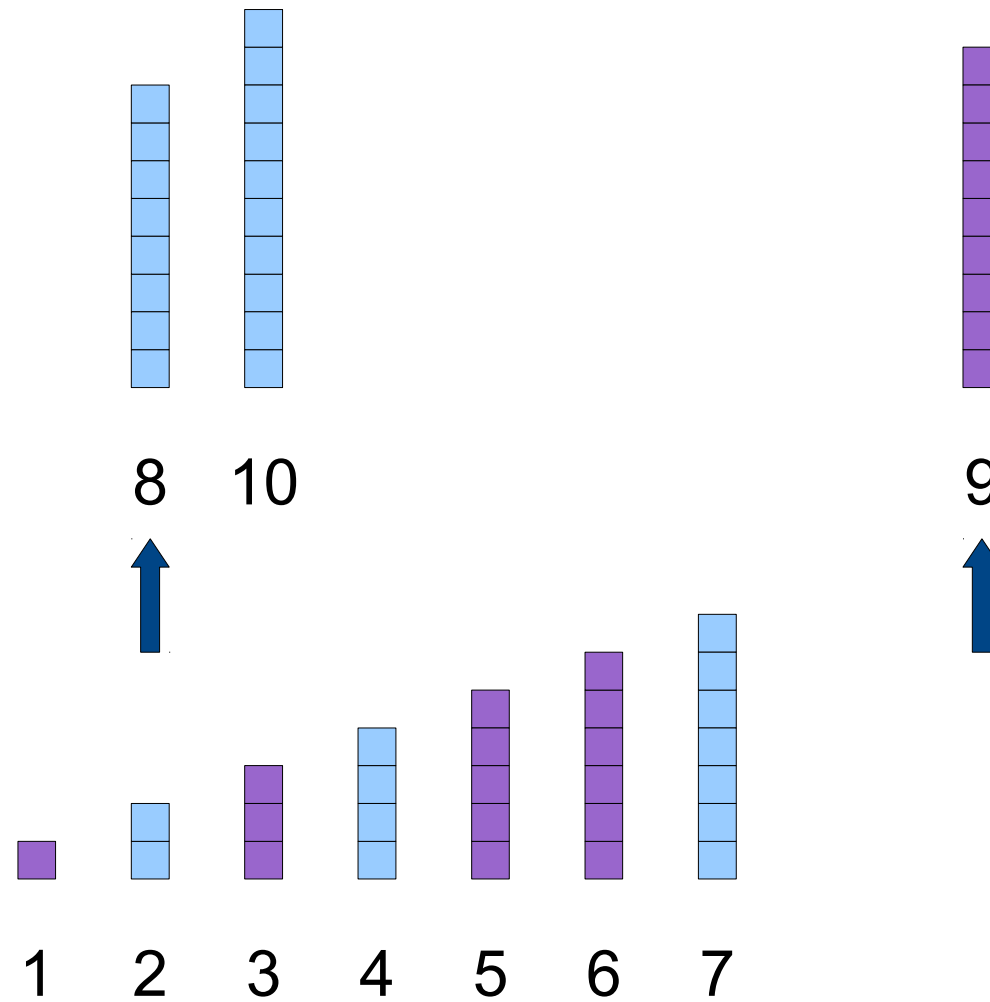


# The Key Insight: *Merge*

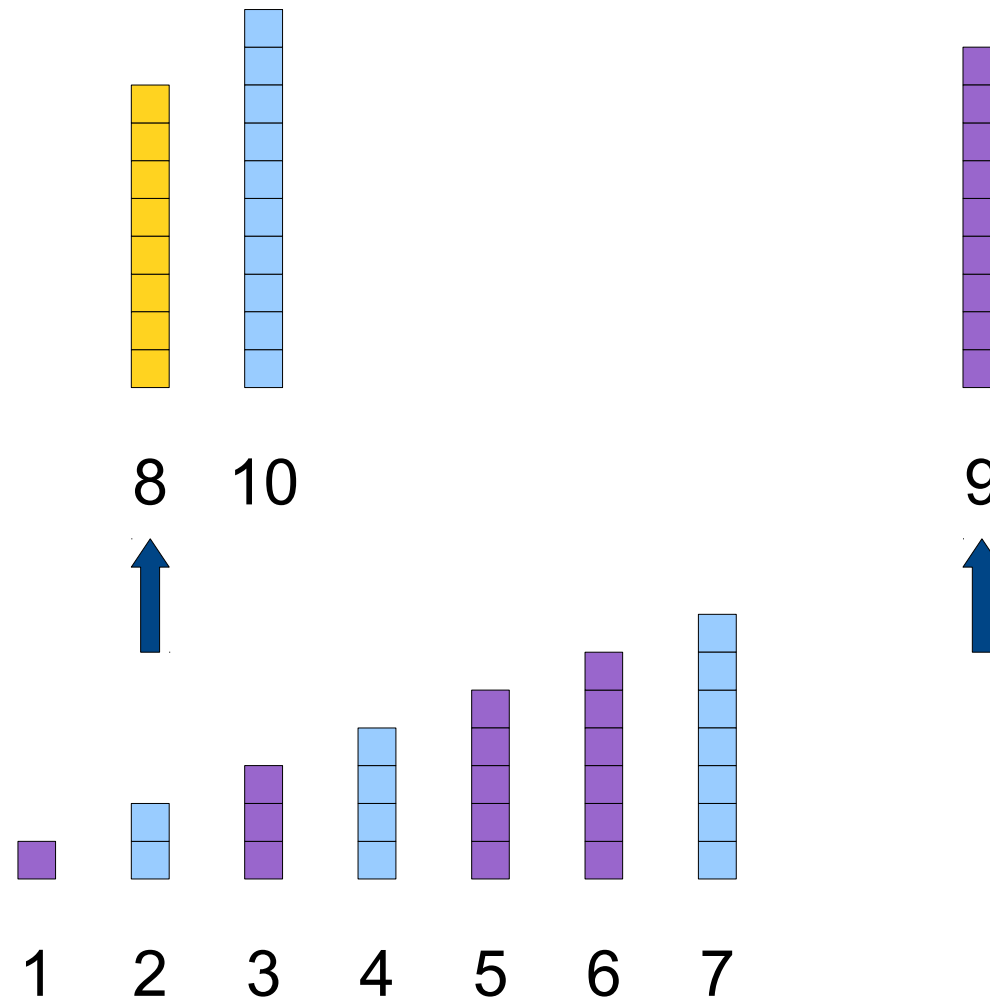




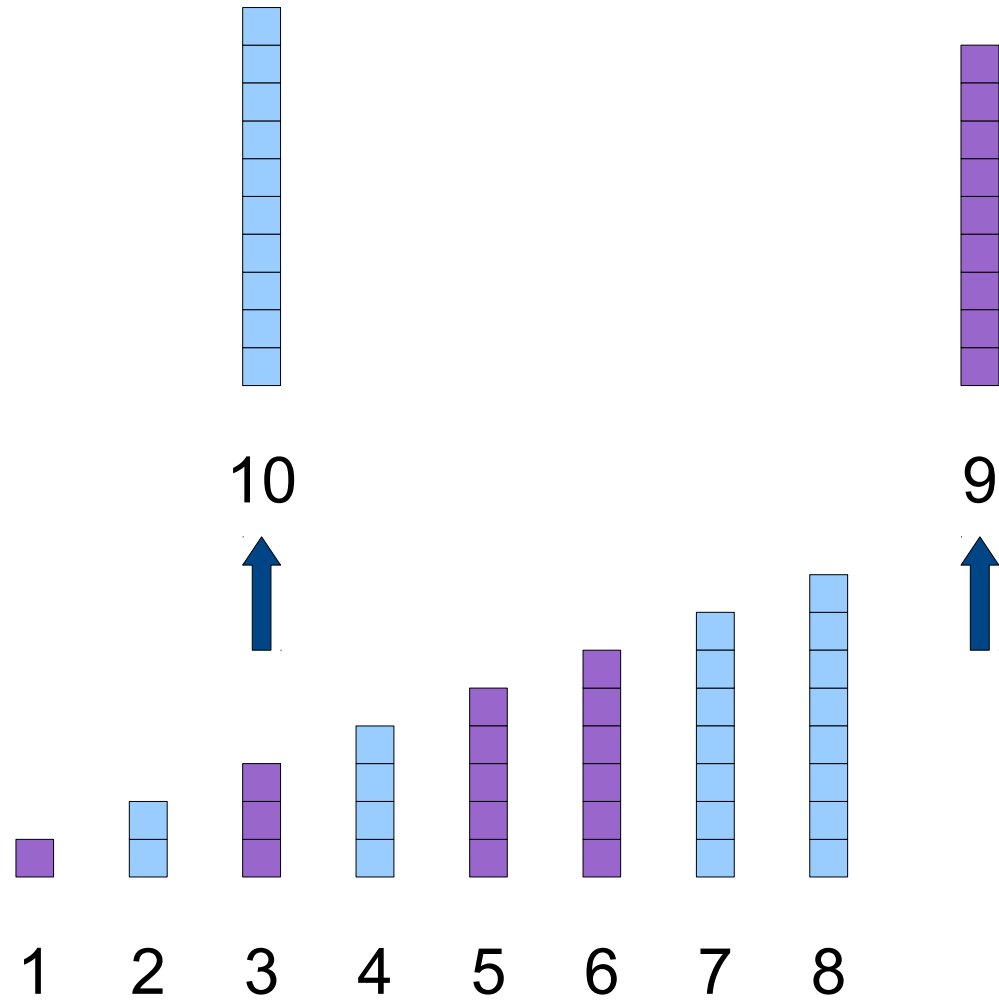
# The Key Insight: *Merge*



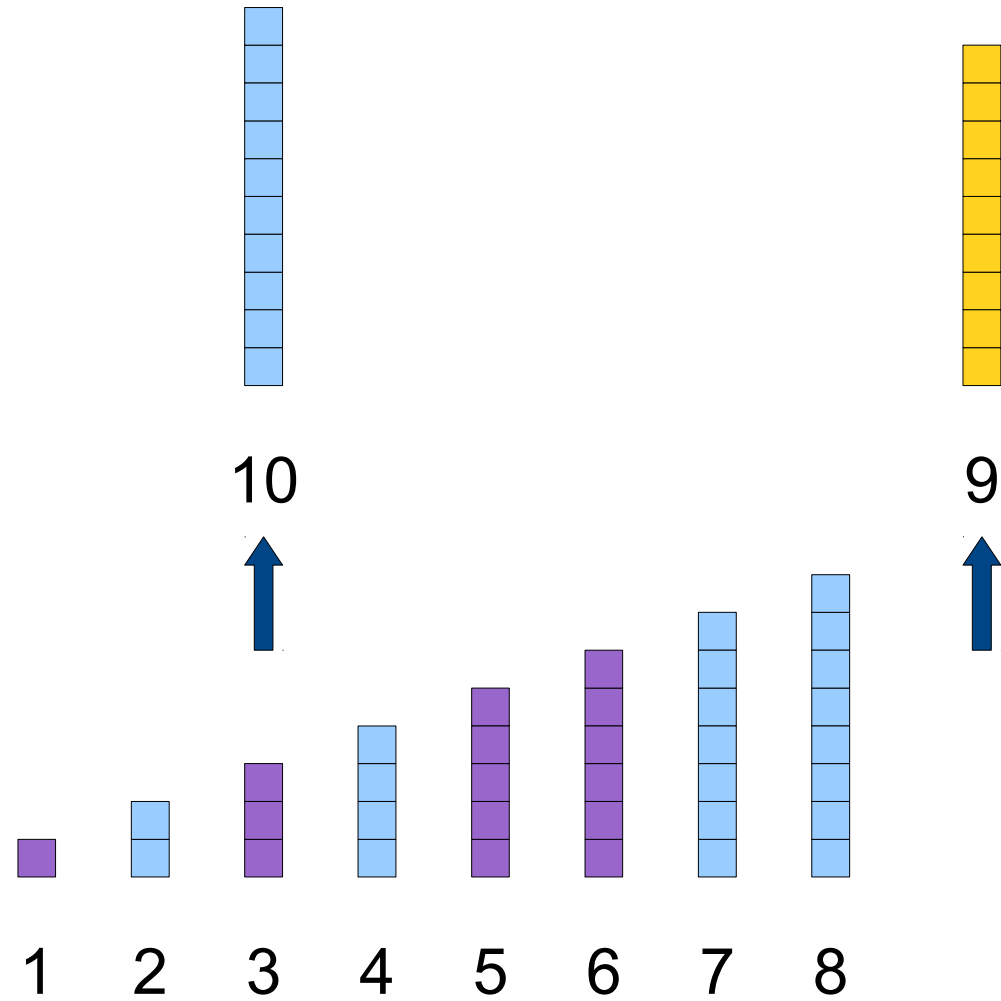
# The Key Insight: *Merge*



# The Key Insight: *Merge*

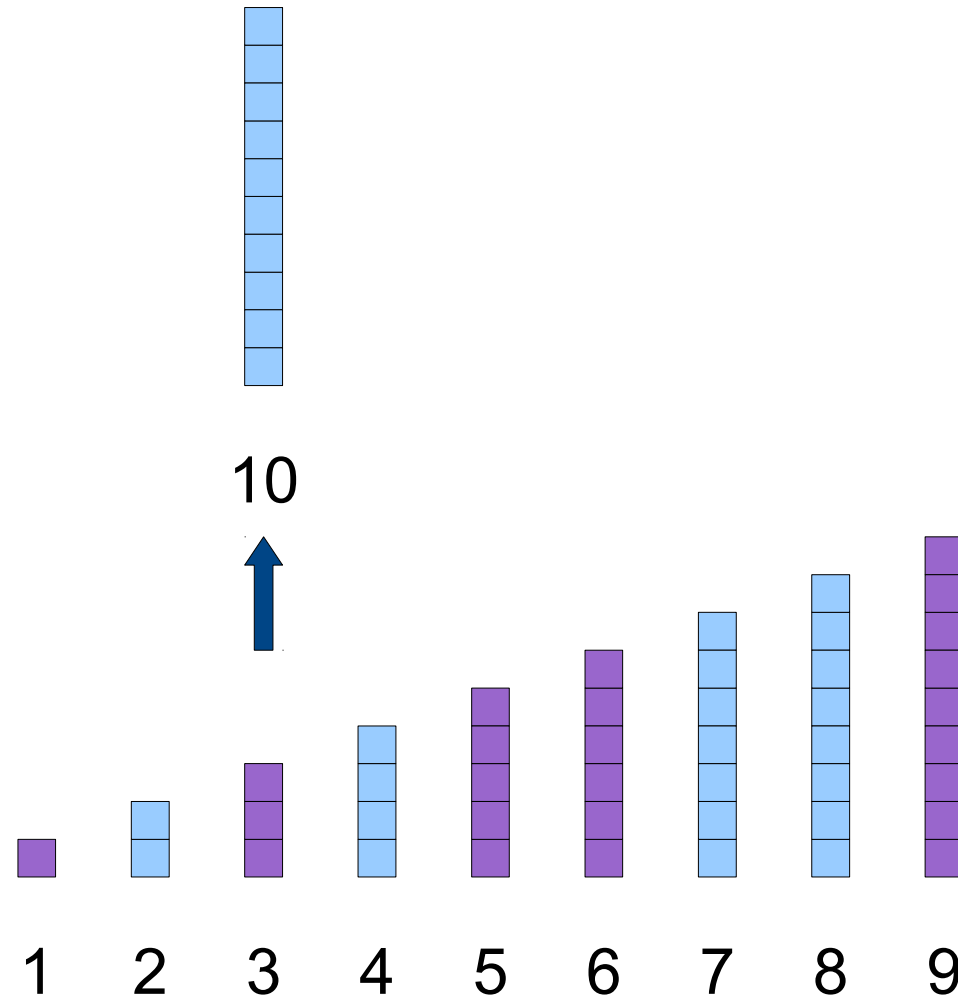


# The Key Insight: *Merge*

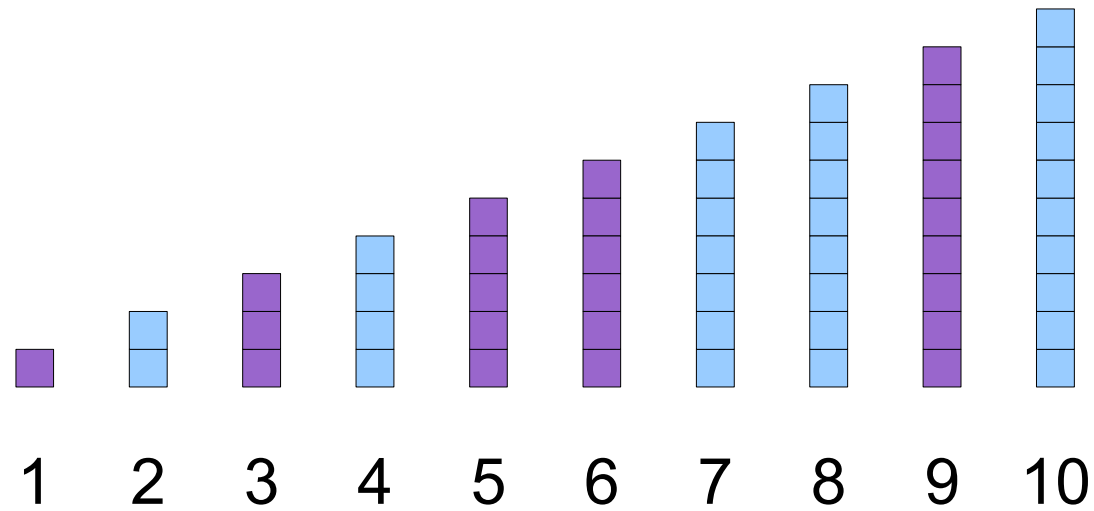




# The Key Insight: *Merge*



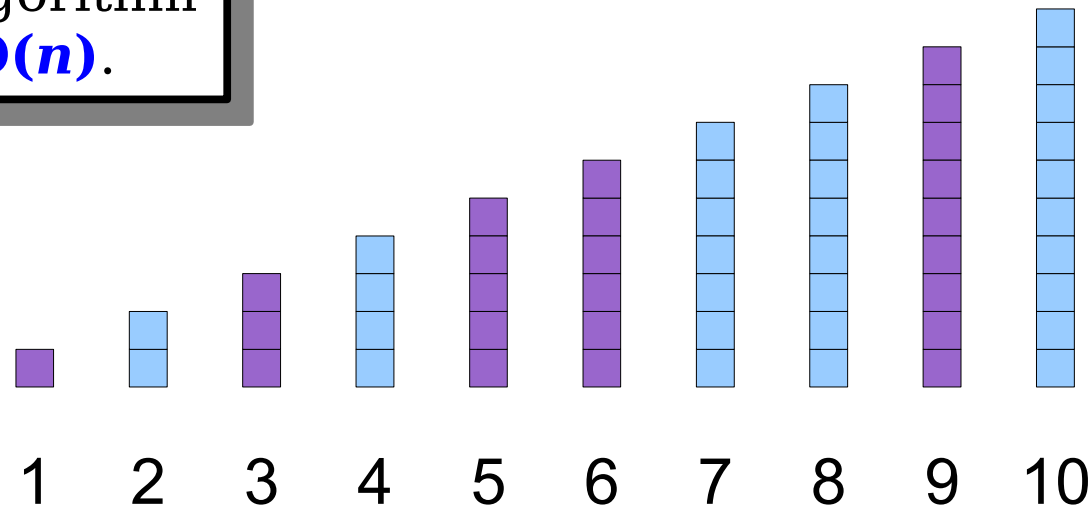
# The Key Insight: *Merge*



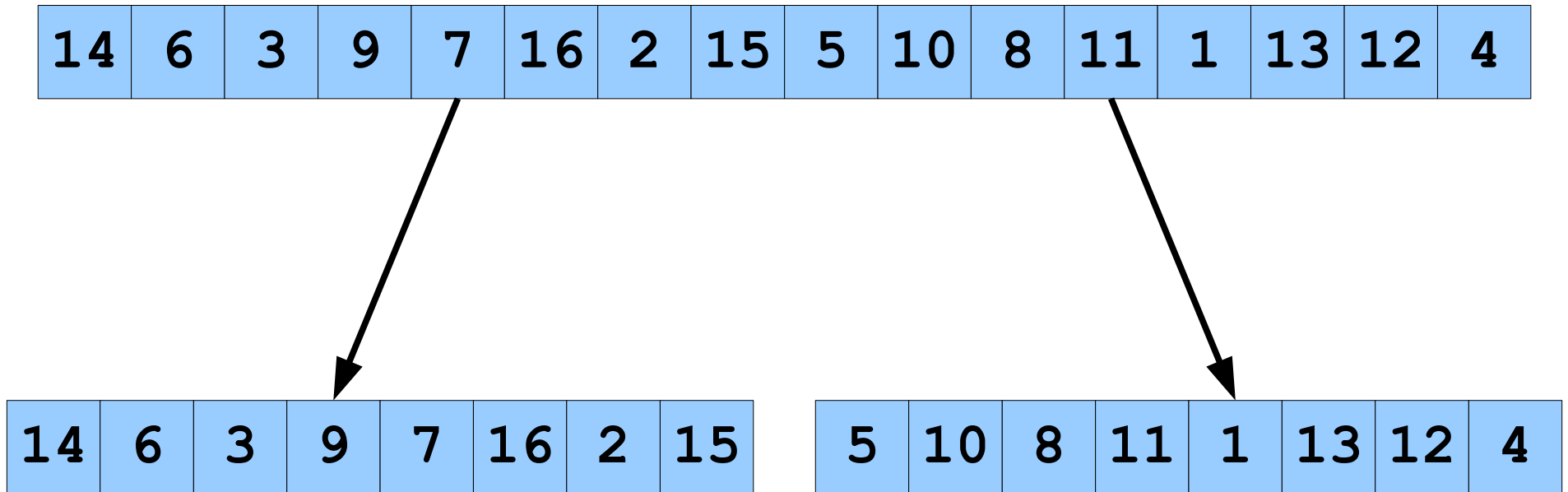
# The Key Insight: *Merge*

Each step makes a single comparison and reduces the number of elements by one.

If there are  $n$  total elements, this algorithm runs in time  $O(n)$ .



# “Split Sort”



1. Split the input in half.

# “Split Sort”

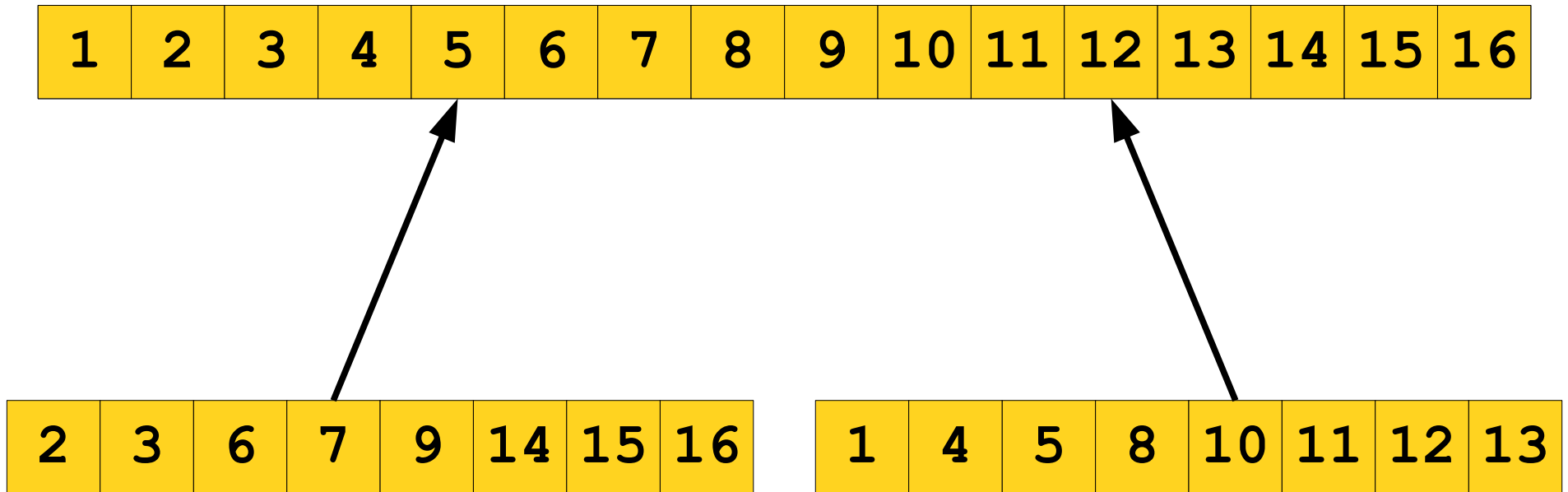
14	6	3	9	7	16	2	15	5	10	8	11	1	13	12	4
----	---	---	---	---	----	---	----	---	----	---	----	---	----	----	---

14	6	3	9	7	16	2	15
----	---	---	---	---	----	---	----

5	10	8	11	1	13	12	4
---	----	---	----	---	----	----	---

1. Split the input in half.
2. Insertion sort each half.

# “Split Sort”



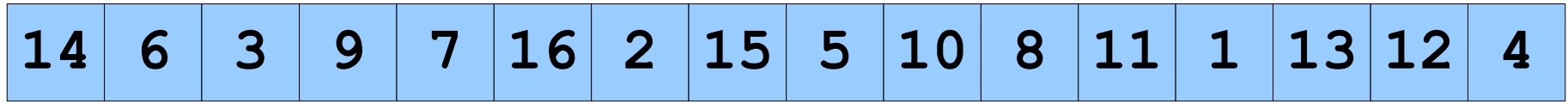
1. Split the input in half.
2. Insertion sort each half.
3. Merge the halves back together.

# “Split Sort”

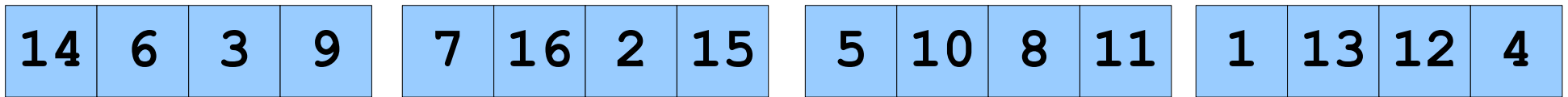
```
void splitSort(Vector<int>& v) {  
    /* Split the vector in half */  
    Vector<int> left, right;  
    for (int i = 0; i < v.size() / 2; i++) {  
        left += v[i];  
    }  
    for (int j = v.size() / 2; j < v.size(); j++) {  
        right += v[j];  
    }  
  
    /* Sort each half. */  
    insertionSort(left);  
    insertionSort(right);  
  
    /* Merge them back together. */  
    merge(left, right, v);  
}
```

**Prediction:** This should be twice as fast as insertion sort.

# “Double Split Sort”



$T(n)$



$T(\frac{1}{4}n)$

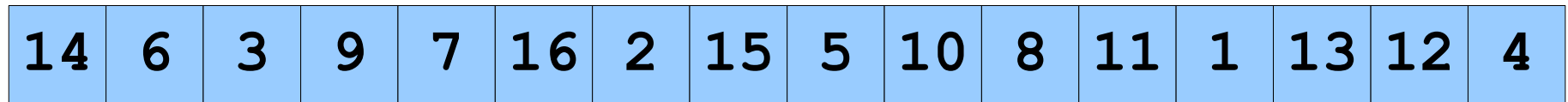
$T(\frac{1}{4}n)$

$T(\frac{1}{4}n)$

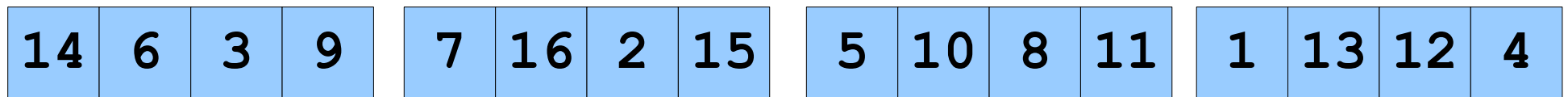
$T(\frac{1}{4}n)$



# “Double Split Sort”



$T(n)$



$\frac{1}{16} T(n)$

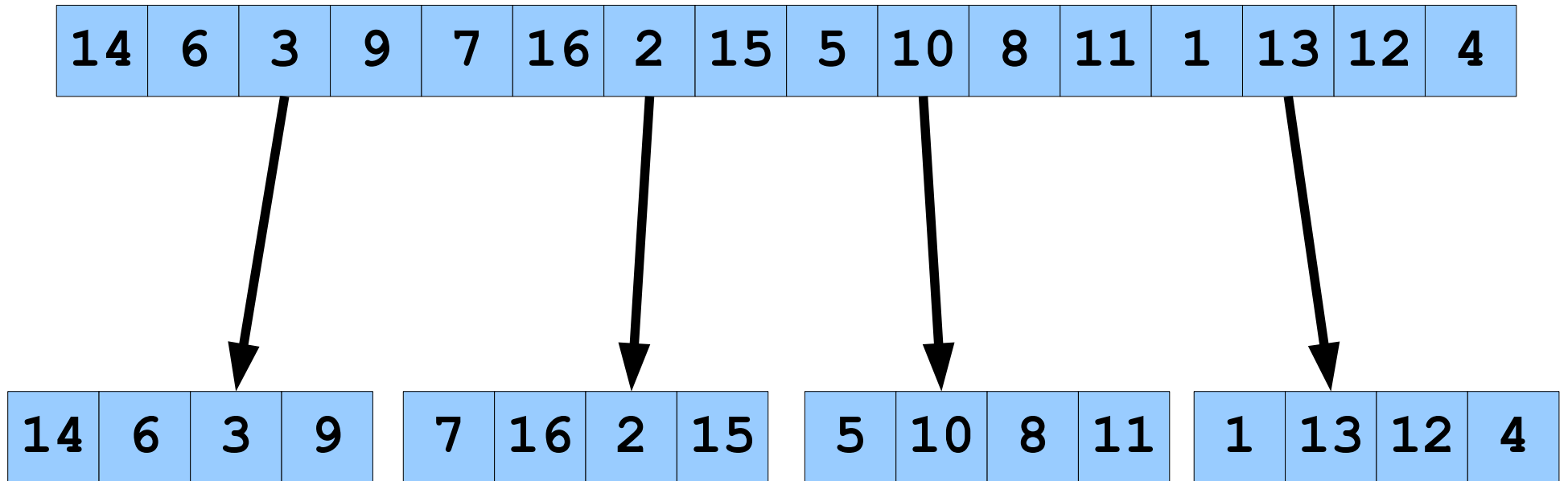
$\frac{1}{16} T(n)$

$\frac{1}{16} T(n)$

$\frac{1}{16} T(n)$

$$4 \cdot \frac{1}{16} T(n) = \frac{1}{4} T(n)$$

# “Double Split Sort”



1. Split the input into quarters.

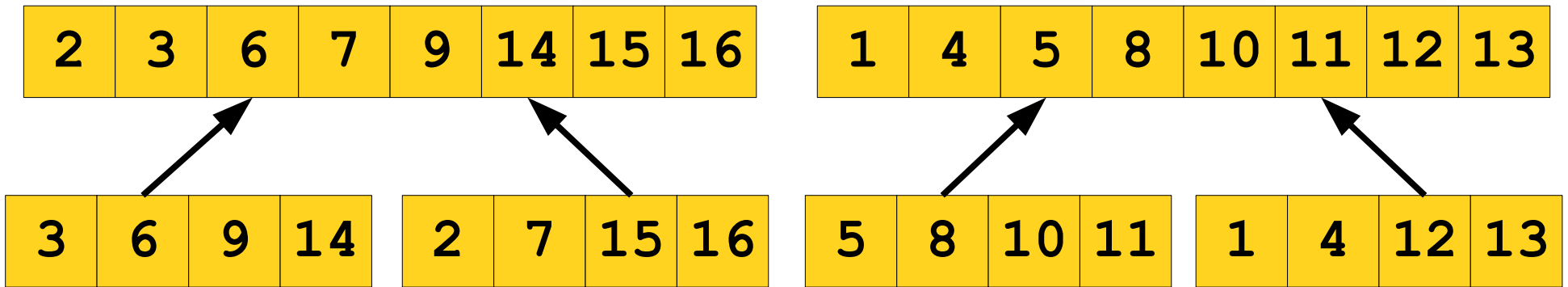
# “Double Split Sort”

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

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

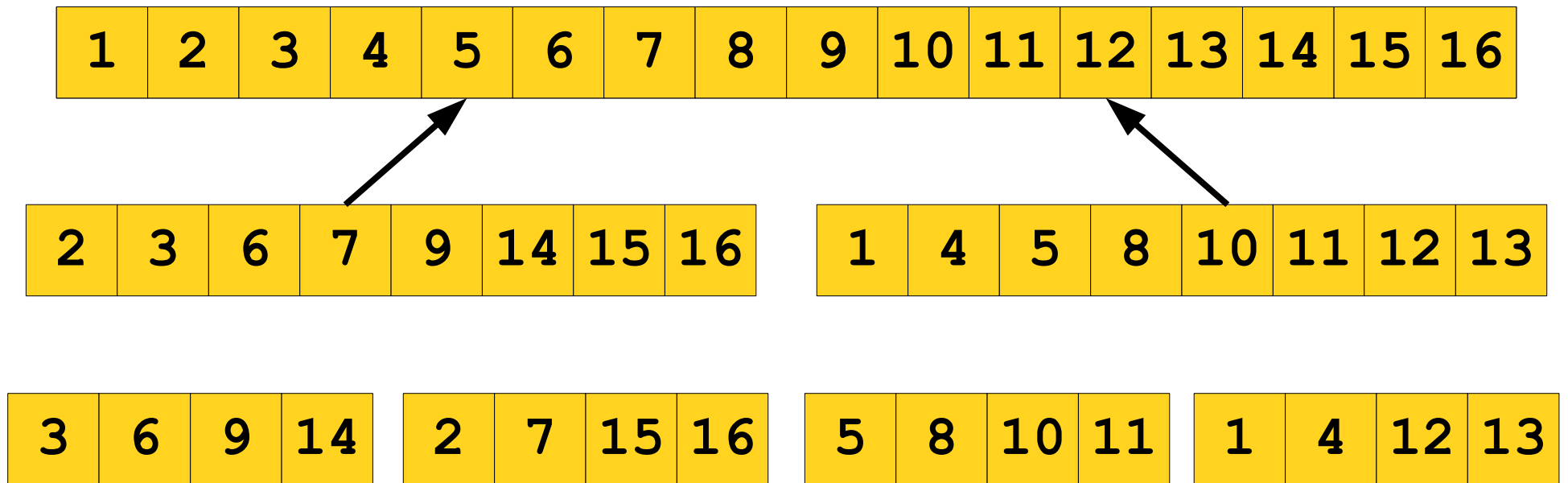
1. Split the input into quarters.
2. Insertion sort each quarter.

# “Double Split Sort”



1. Split the input into quarters.
2. Insertion sort each quarter.
3. Merge two pairs of quarters into halves.

# “Double Split Sort”



1. Split the input into quarters.
2. Insertion sort each quarter.
3. Merge two pairs of quarters into halves.
4. Merge the two halves back together.

**Prediction:** This should be four times as fast as insertion sort.

**Time-Out for Announcements!**

# Assignment 4

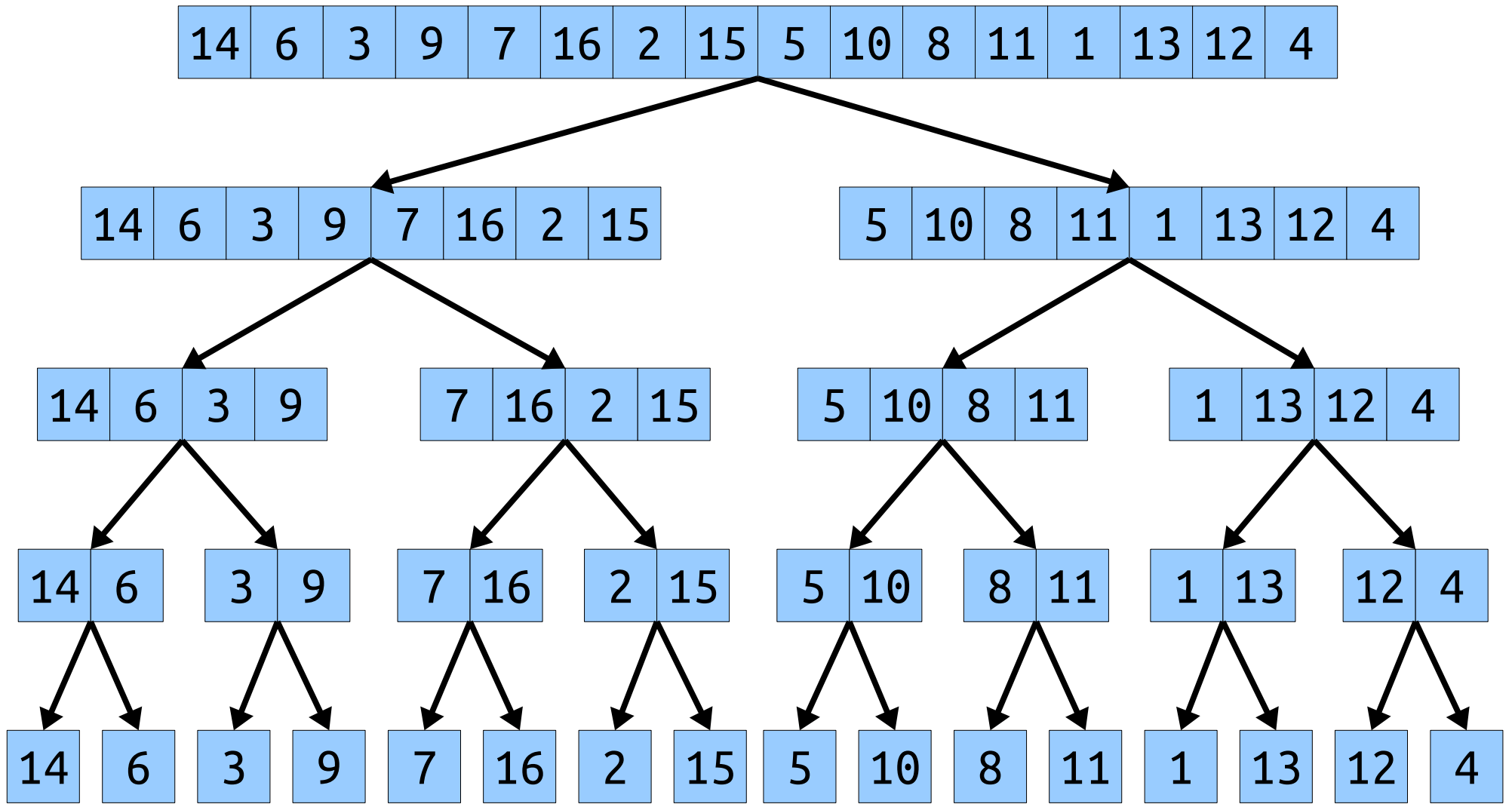
- Assignment 4 (***Recursion to the Rescue***) goes out today. It's a three-parter designed to give you a sense of just how powerful recursion is.
- You are encouraged to work in pairs on this one.
- We recommend making slow, steady progress on this assignment. There's a suggested timeline on the front of the handout.
- YEAH Hours are tonight at 7PM in 380-380Y.

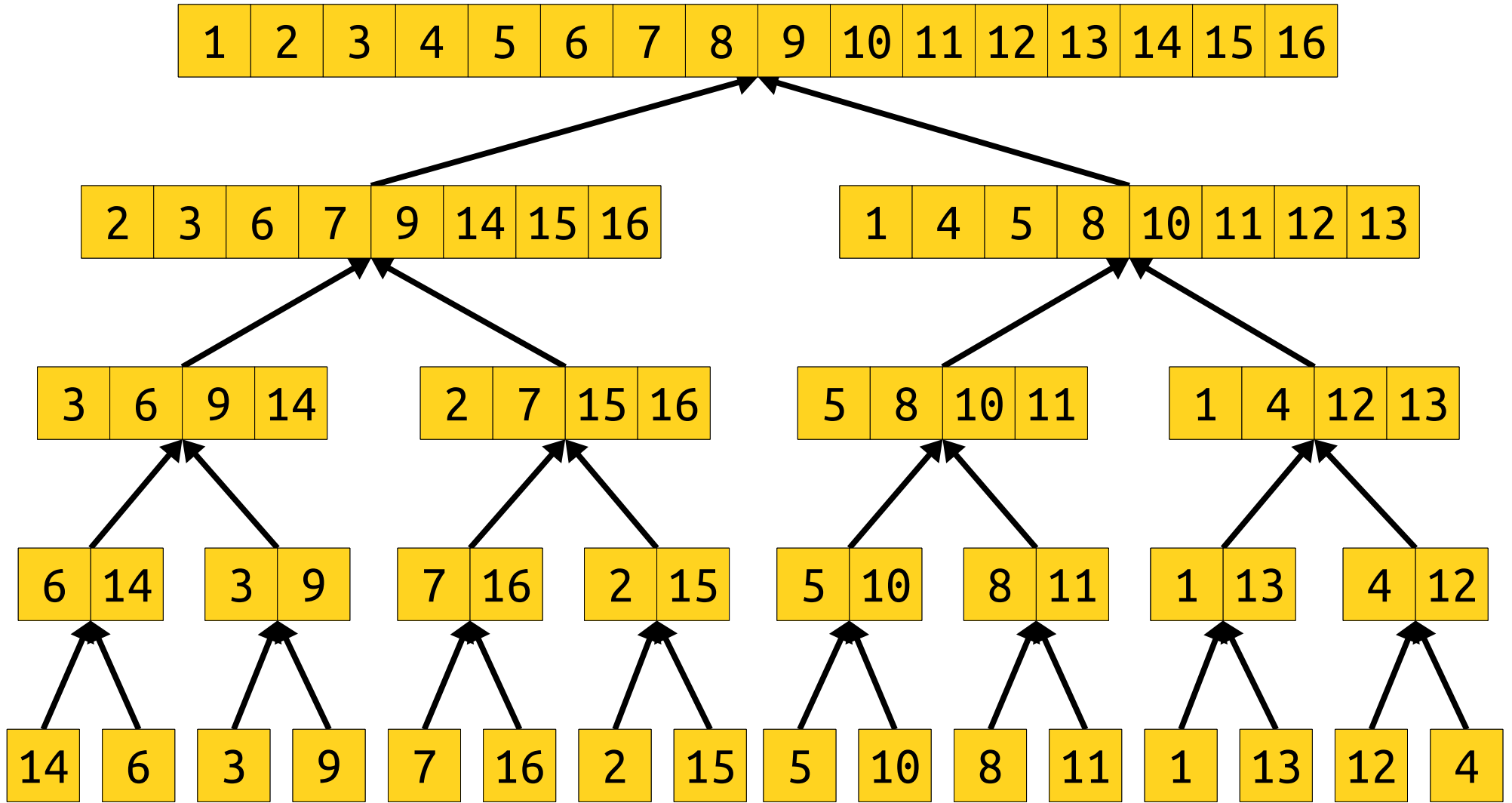
More Assorted Sorts of Sorts!



# Splitting to the Extreme

- Splitting our array in half, sorting each half, and merging the halves was twice as fast as insertion sort.
- Splitting our array in quarters, sorting each quarter, and merging the quarters was four times as fast as insertion sort.
- **Question:** What happens if we *never stop splitting*?





# Mergesort

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

```
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are
     * already sorted.
     */
    if (v.size() <= 1) return;

    /* Split v into two subvectors. */
    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++) {
        left += v[i];
    }
    for (int i = v.size() / 2; i < v.size(); i++) {
        right += v[i];
    }

    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

    /* Combine them together. */
    merge(left, right, v);
}
```

How fast is mergesort?

First, the numbers.

Now, the theory!



```
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are
     * already sorted.
     */
    if (v.size() <= 1) return;

    /* Split v into two subvectors. */
    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++) {
        left += v[i];
    }
    for (int i = v.size() / 2; i < v.size(); i++) {
        right += v[i];
    }

    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

    /* Combine them together. */
    merge(left, right, v);
}
```

```

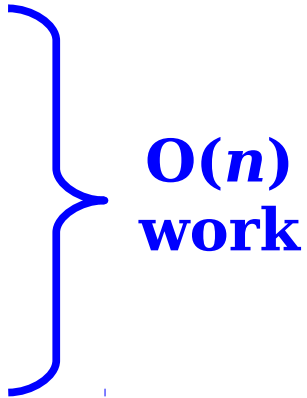
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are
     * already sorted.
     */
    if (v.size() <= 1) return;


    /* Split v into two subvectors. */
    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++) {
        left += v[i];
    }
    for (int i = v.size() / 2; i < v.size(); i++) {
        right += v[i];
    }

    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

    /* Combine them together. */
    merge(left, right, v);
}

```


**O(n)  
work**

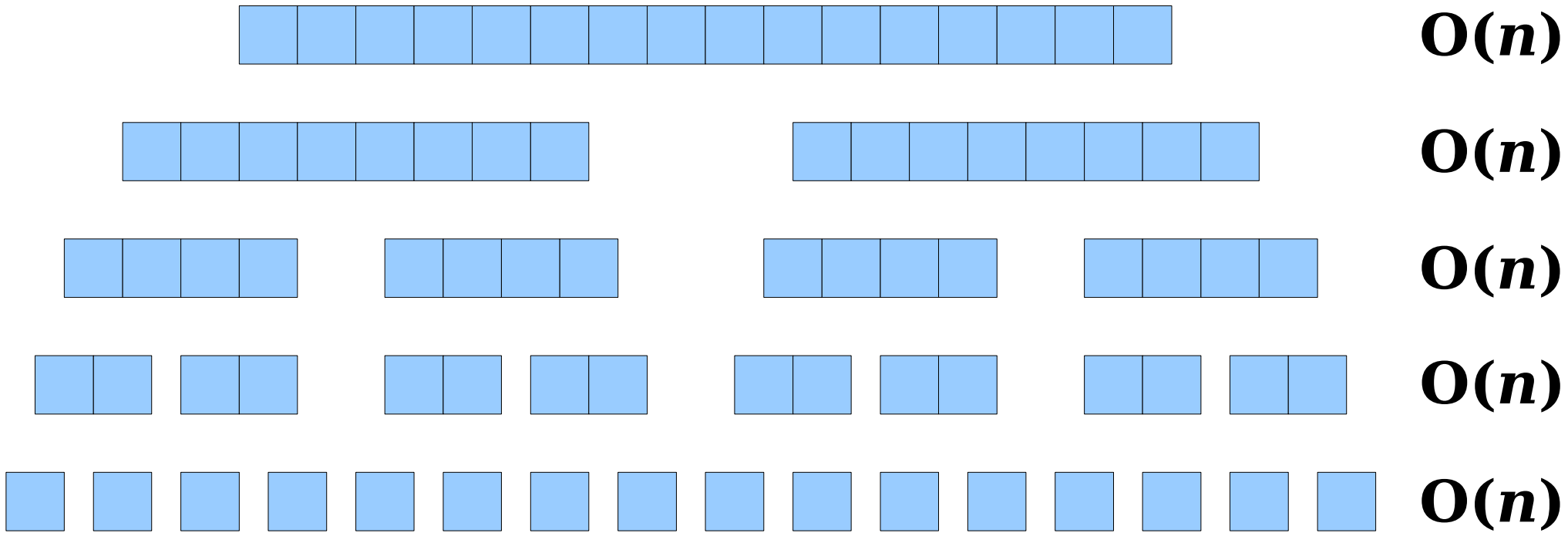

**O(n)  
work**

```
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are
     * already sorted.
     */
    if (v.size() <= 1) return;

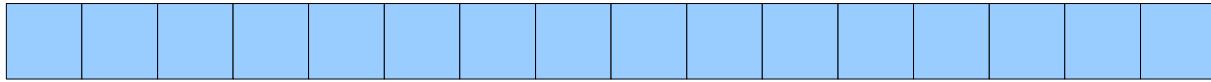
    /* Split v into two subvectors. */
    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++) {
        left += v[i];
    }
    for (int i = v.size() / 2; i < v.size(); i++) {
        right += v[i];
    }

    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

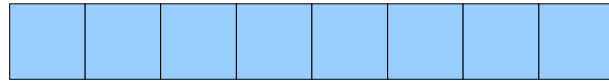
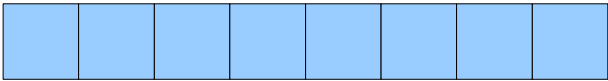
    /* Combine them together. */
    merge(left, right, v);
}
```



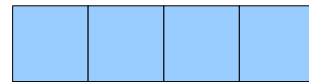
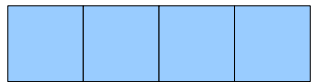
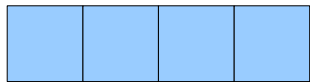
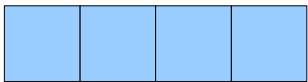
How much work does mergesort do at each level of recursion?



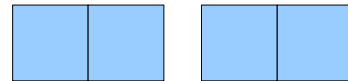
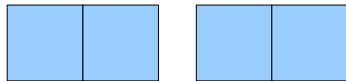
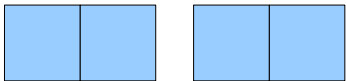
**$O(n)$**



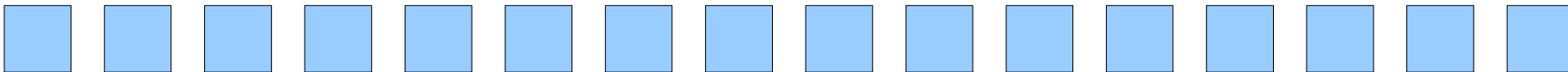
**$O(n)$**



**$O(n)$**

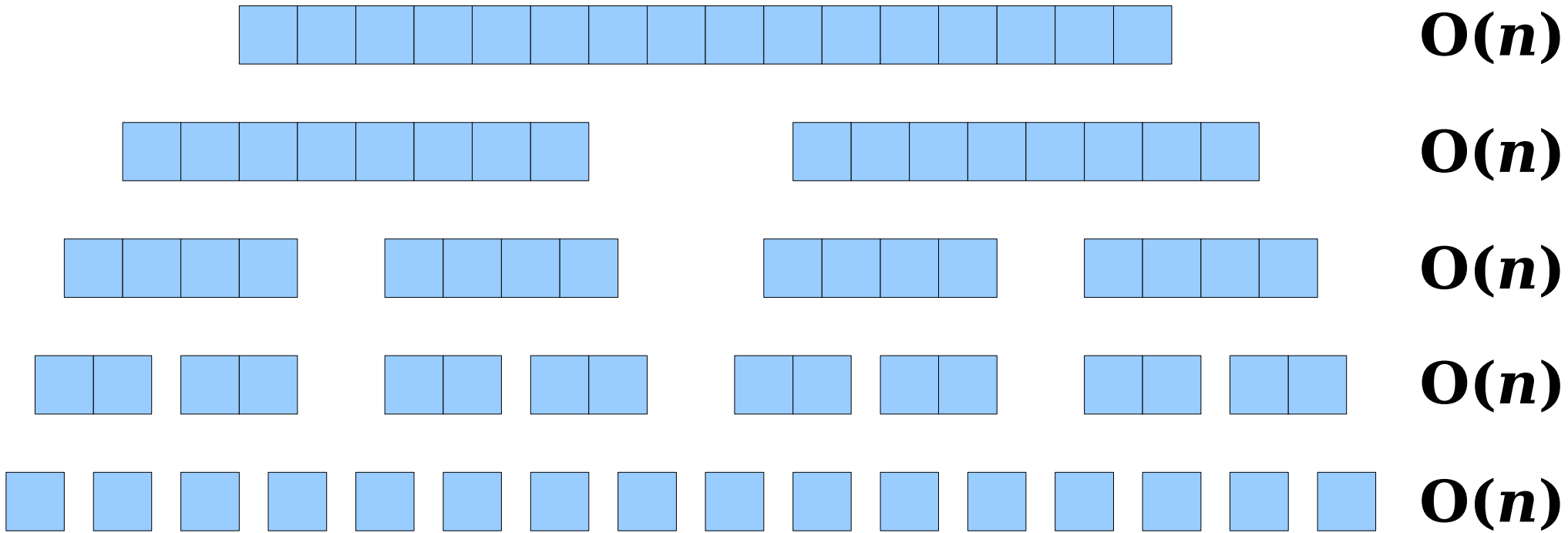


**$O(n)$**

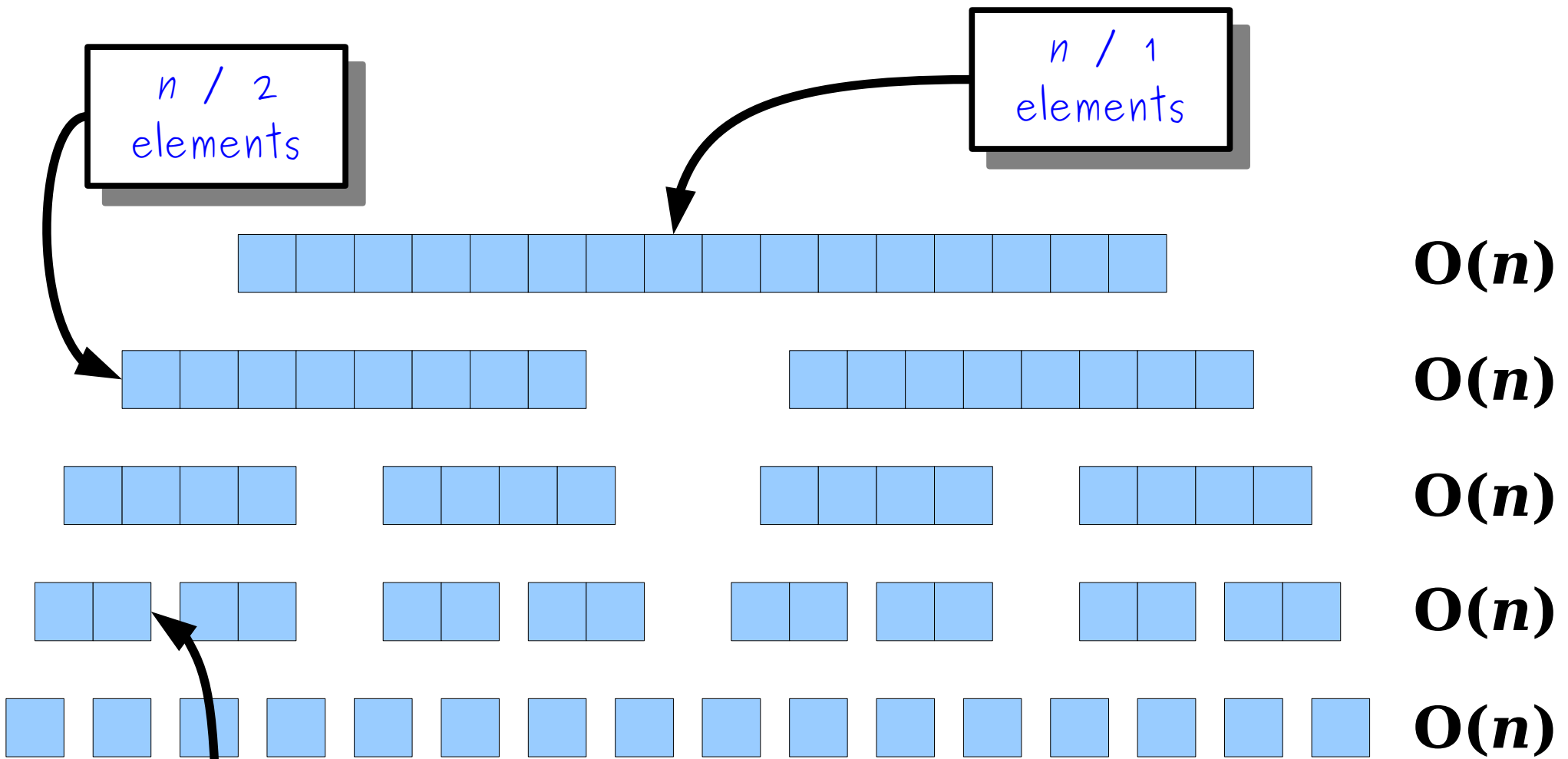


**$O(n)$**

How many levels are there?

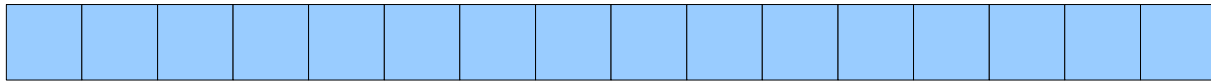


Each recursive call cuts the array size in half.

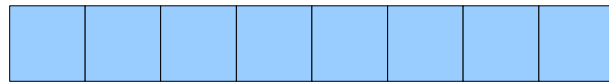
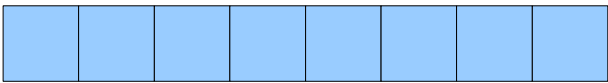


$n / 8$   
 elements

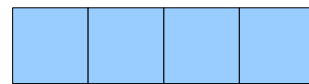
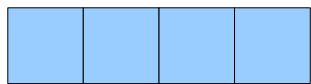
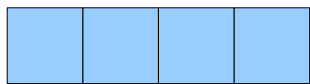
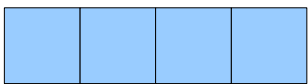
After  $k$  layers of the recursion,  
 if the original array has size  $n$ ,  
 each subarray has size  $n / 2^k$ .



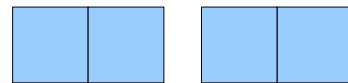
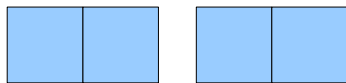
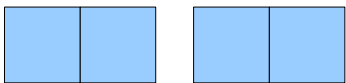
**$O(n)$**



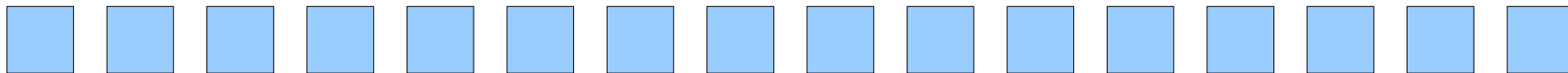
**$O(n)$**



**$O(n)$**



**$O(n)$**



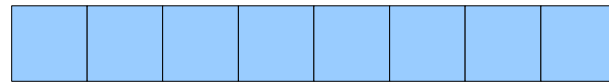
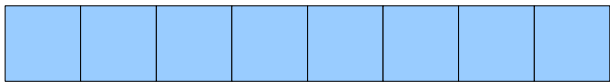
**$O(n)$**

The recursion stops when  
we're down to a single  
element.

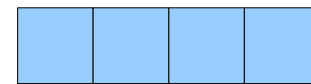
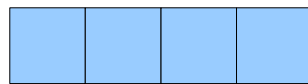
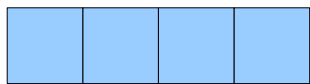
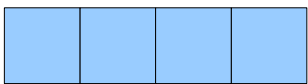




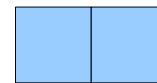
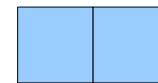
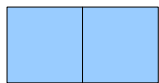
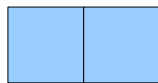
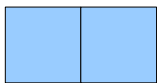
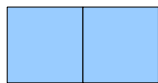
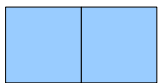
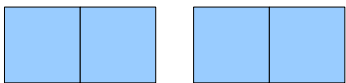
**$O(n)$**



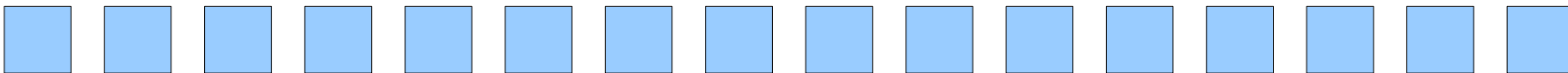
**$O(n)$**



**$O(n)$**



**$O(n)$**



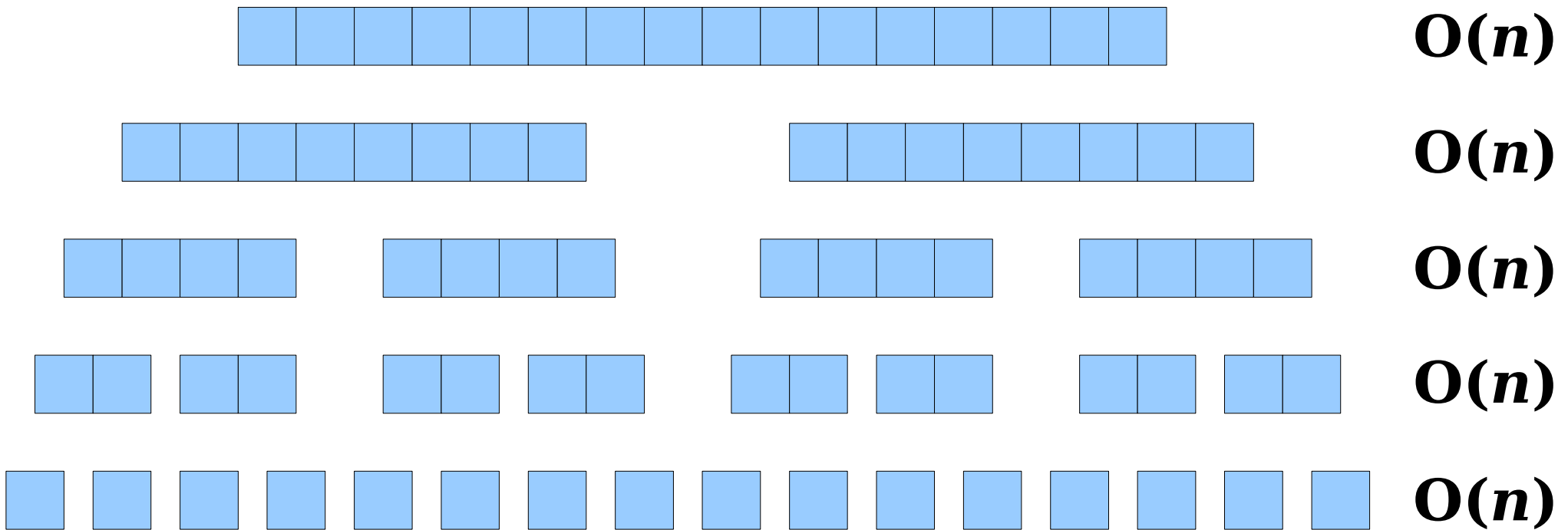
**$O(n)$**

***Useful intuition:***

you can only cut something in half  $O(\log n)$  times before you run out of elements.

What choice of  $k$  makes  $n / 2^k = 1$ ?

***Answer:***  $k = \log_2 n$ .



There are  $O(\log n)$  levels in the recursion.

Each level does  $O(n)$  work.

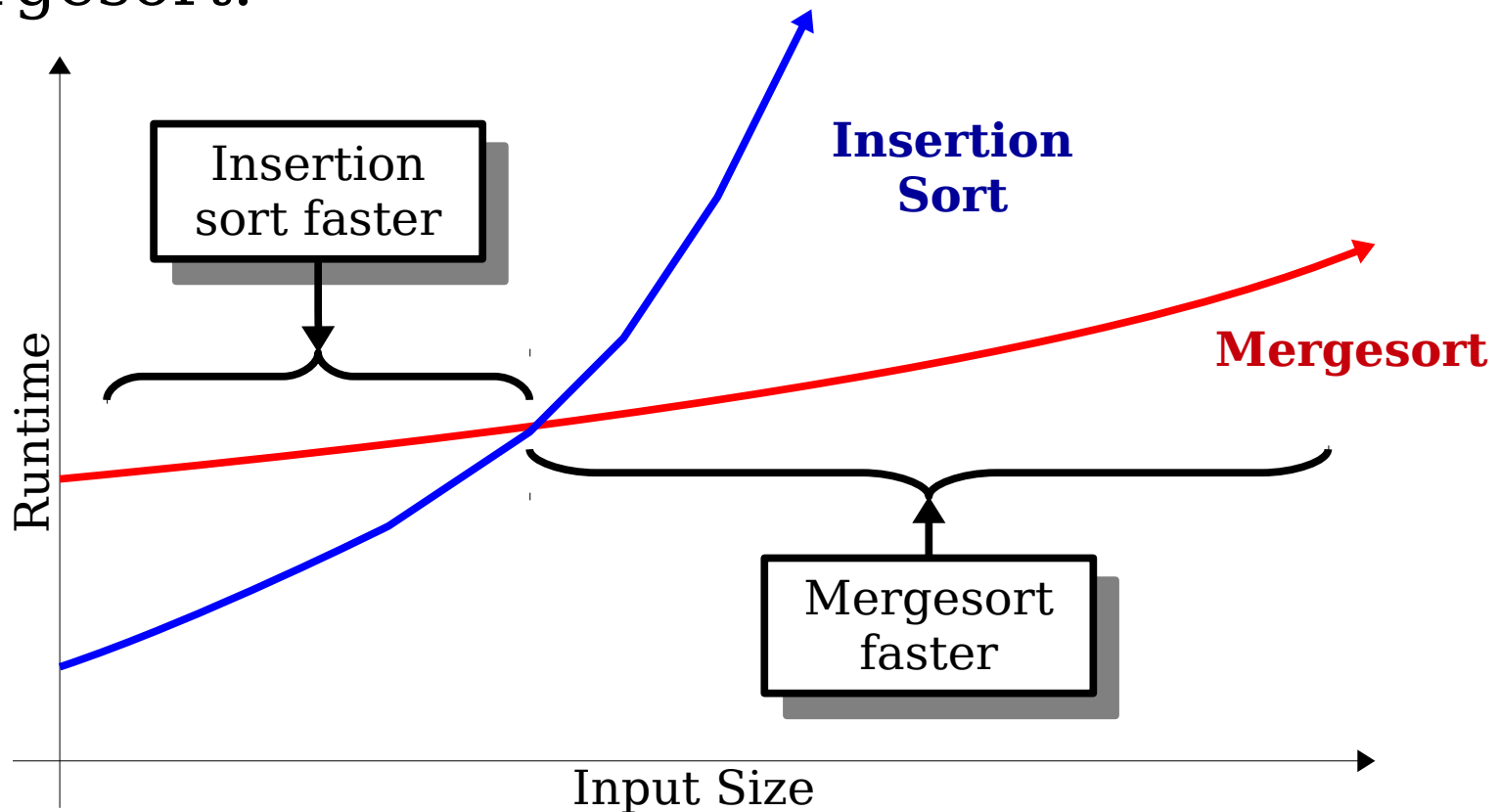
Total work done:  **$O(n \log n)$** .

# Can we do Better?

- Mergesort runs in time  $O(n \log n)$ , which is faster than insertion sort's  $O(n^2)$ .
- Can we do better than this?
  - In general, **no**: comparison-based sorts cannot have a worst-case runtime better than  $O(n \log n)$ .
- ***In the worst case, we can only get faster by a constant factor!***

# An Interesting Observation

- Big-O notation talks about long-term growth, but says nothing about small inputs.
- For small inputs, insertion sort can be faster than mergesort.



# Hybrid Mergesort

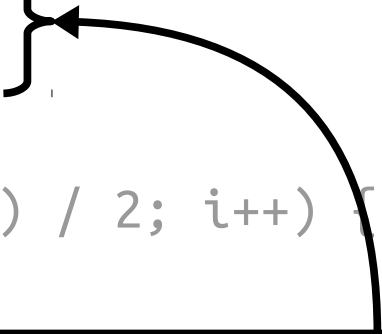
```
void hybridMergesort(Vector<int>& v) {
    if (v.size() <= kCutoffSize) {
        insertionSort(v);
    } else {
        Vector<int> left, right;
        for (int i = 0; i < v.size() / 2; i++) {
            left += v[i];
        }
        for (int i = v.size() / 2; i < v.size(); i++) {
            right += v[i];
        }

        hybridMergesort(left);
        hybridMergesort(right);

        merge(left, right, v);
    }
}
```

# Hybrid Mergesort

```
void hybridMergesort(Vector<int>& v) {  
    if (v.size() <= kCutoffSize) {  
        insertionSort(v);  
    } else {  
        Vector<int> left, right;  
        for (int i = 0; i < v.size() / 2; i++) {  
            left += v[i];  
        }  
        for (int i = v.size() / 2;  
            i < v.size(); i++) {  
            right += v[i];  
        }  
  
        hybridMergesort(left);  
        hybridMergesort(right);  
  
        merge(left, right, v);  
    }  
}
```



Use insertion sort for small inputs where insertion sort is faster than mergesort.

**Question to ponder:** How would you determine the value of `kCutoffSize` to use?

Closing the Loop

```
bool linearSearch(const string& str, char ch) {  
    for (int i = 0; i < str.length(); i++) {  
        if (str[i] == ch) {  
            return true;  
        }  
    }  
    return false;  
}
```

Best-Case Runtime:  **$O(1)$**

Worst-Case Runtime:  **$O(n)$**

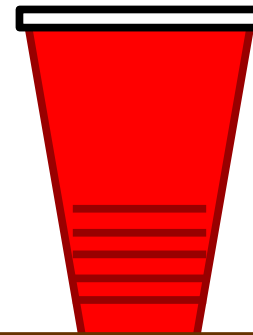
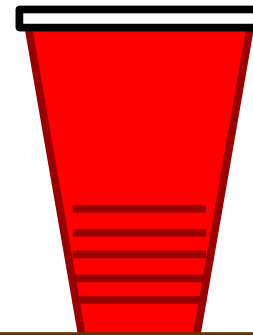
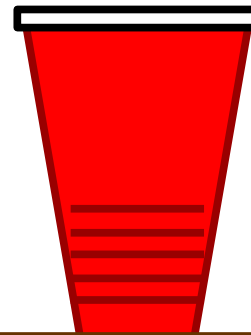
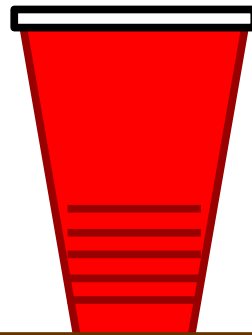
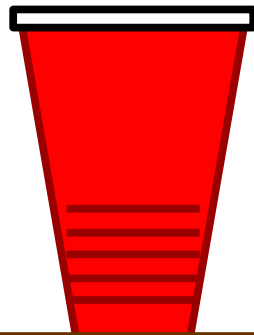
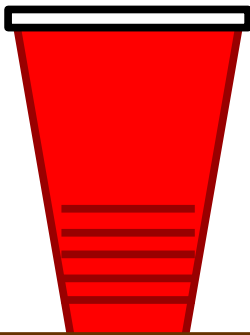
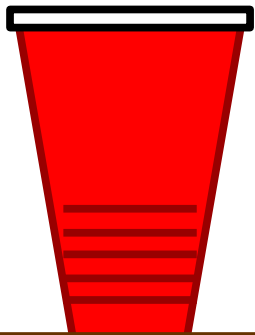
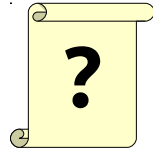
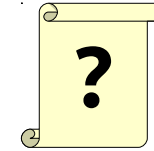
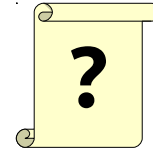
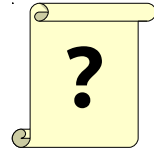
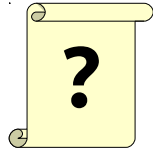
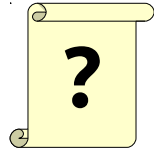
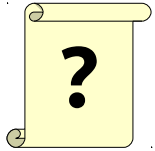


Suppose we want to search an array for an element, and we know that array is sorted.

Can we do better than linear search?

Each cup  
contains a  
number.

Numbers are  
sorted from left  
to right



Are any of  
these numbers  
equal to 106?

Each cup contains a number.

Numbers are sorted from left to right

137

Can 106 be here?

Or here?

Or here?



Are any of these numbers equal to 106?

Each cup  
contains a  
number.

Numbers are  
sorted from left  
to right

96

Can 106  
be here?

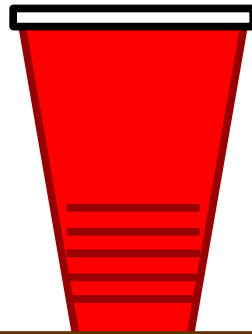


Are any of  
these numbers  
equal to 106?

Each cup  
contains a  
number.

Numbers are  
sorted from left  
to right

103



Are any of  
these numbers  
equal to 106?

Each cup  
contains a  
number.

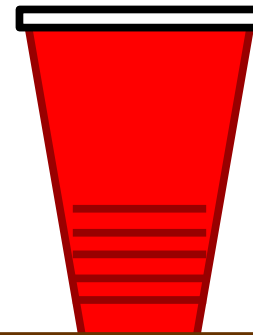
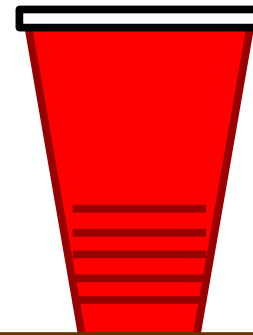
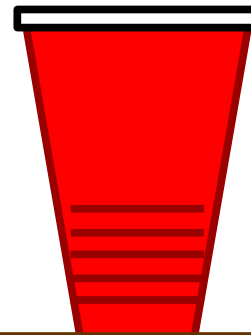
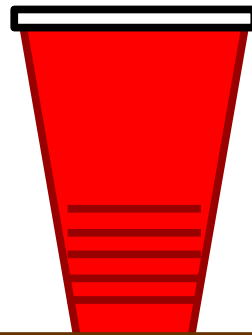
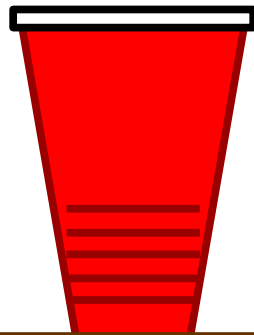
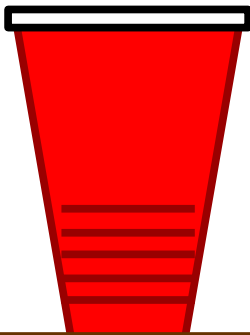
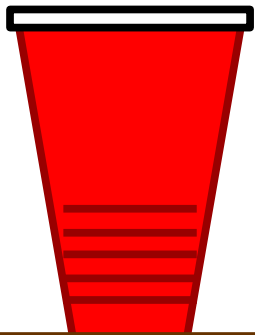
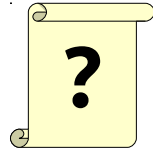
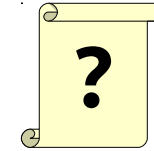
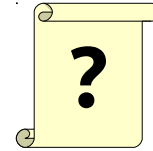
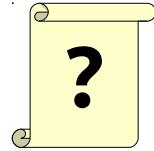
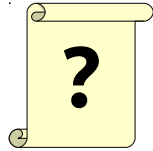
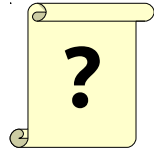
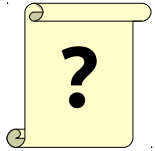
Numbers are  
sorted from left  
to right

Alas, 106 is not to be found here.

Are any of  
these numbers  
equal to 106?

Each cup  
contains a  
number.

Numbers are  
sorted from left  
to right



Are any of  
these numbers  
equal to 106?

Each cup contains a number.

Numbers are sorted from left to right

101

Or here?

Or here?

Can 106 be here?



Are any of these numbers equal to 106?

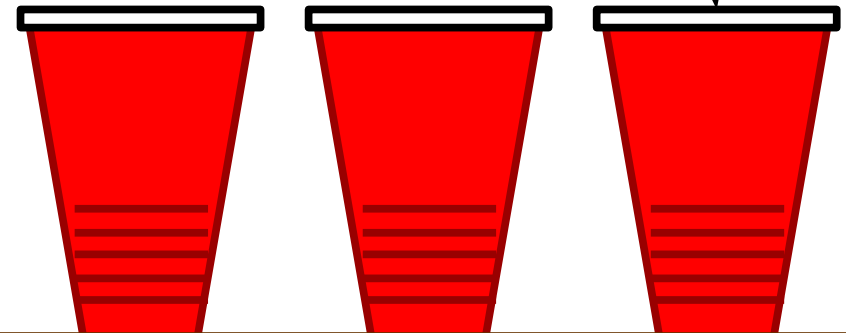


Each cup  
contains a  
number.

Numbers are  
sorted from left  
to right

109

Can 106  
be here?



Are any of  
these numbers  
equal to 106?

Each cup  
contains a  
number.

Numbers are  
sorted from left  
to right

106

Are any of  
these numbers  
equal to 106?

*Thanks to former head TA Dawson Zhou for this idea! Except he did it IRL.*

This algorithm is called ***binary search***.

```

bool binarySearchRec(const Vector<int>& elems, int key,
                    int low, int high) {
    /* Base case: If we're out of elements, horror of horrors!
     * Our element does not exist.
     */
    if (low == high) return false;

    /* Probe the middle element. */
    int mid = low + (high - low) / 2;

    /* We might find what we're looking for! */
    if (key == elems[mid]) return true;

    /* Otherwise, discard half the elements and search
     * the appropriate section.
     */
    if (key < elems[mid]) {
        return binarySearchRec(elems, key, low, mid);
    } else {
        return binarySearchRec(elems, key, mid + 1, high);
    }
}

```

***Question to ponder:***

how does this code  
correspond to the  
example from earlier?

```

bool binarySearch(const Vector<int>& elems, int key) {
    return binarySearchRec(elems, key, 0, elems.size());
}

```

# Binary Search

- How fast is binary search?
  - Each round does a constant amount of work (checking how the key relates to the middle).
  - Each round tosses away half the elements.
  - We can only toss away half the elements  $O(\log n)$  times before no elements are left.
  - Worst-case runtime:  **$O(\log n)$** .
  - Question to ponder: what's the best-case runtime?
- This is *exponentially* faster than linear search!

# Why All This Matters

- Big-O notation gives us a ***quantitative way*** to predict runtimes.
- Those predictions provide a ***quantitative intuition*** for how to improve our algorithms.
- Understanding the nuances of big-O notation then leads us to design algorithms that are better than the sum of their parts.
- We can use ***binary search*** to look inside sorted sequences really, really quickly.

# Your Action Items

- ***Start Assignment 4***
  - You have plenty of time to complete this assignment. Starting early will give you plenty of time to think things through.
- ***Read Chapter 10 of the Textbook***
  - There's a bunch of goodies in there we didn't have time to explore here.

# Next Time

- ***Designing Abstractions***
  - How do you build new container classes?
- ***Class Design***
  - What do classes look like in C++?