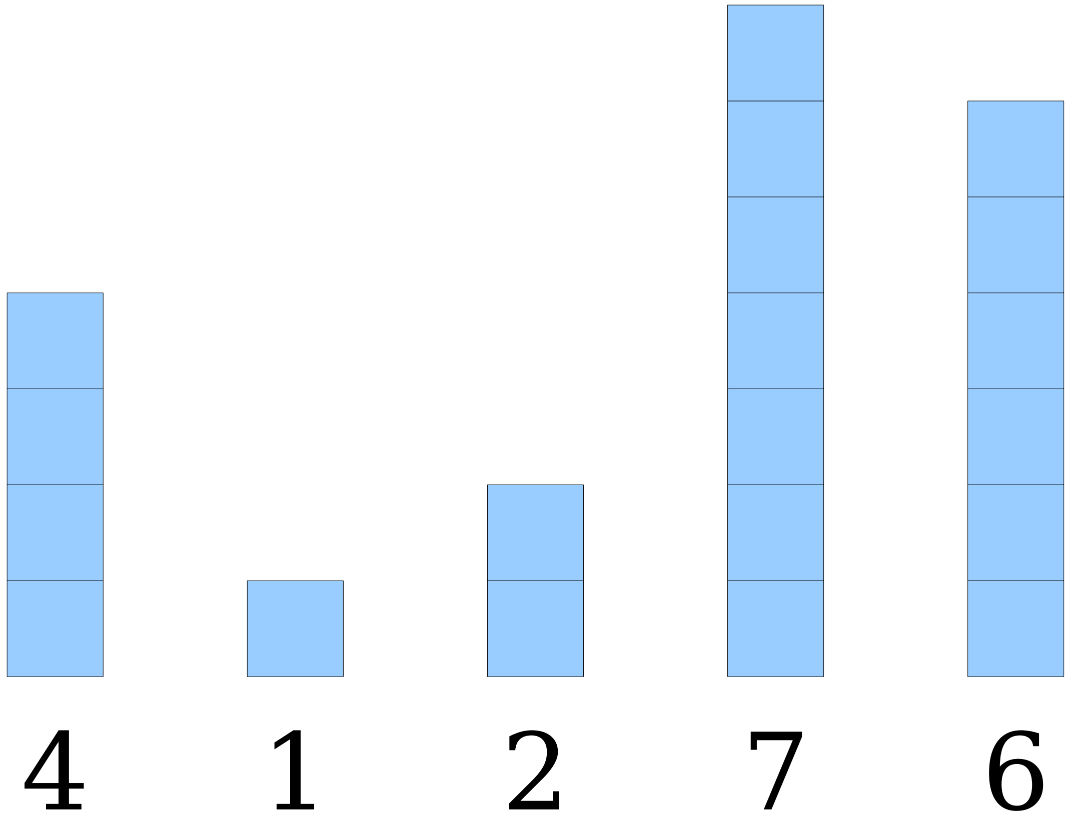
Searching and Sorting

Part Two

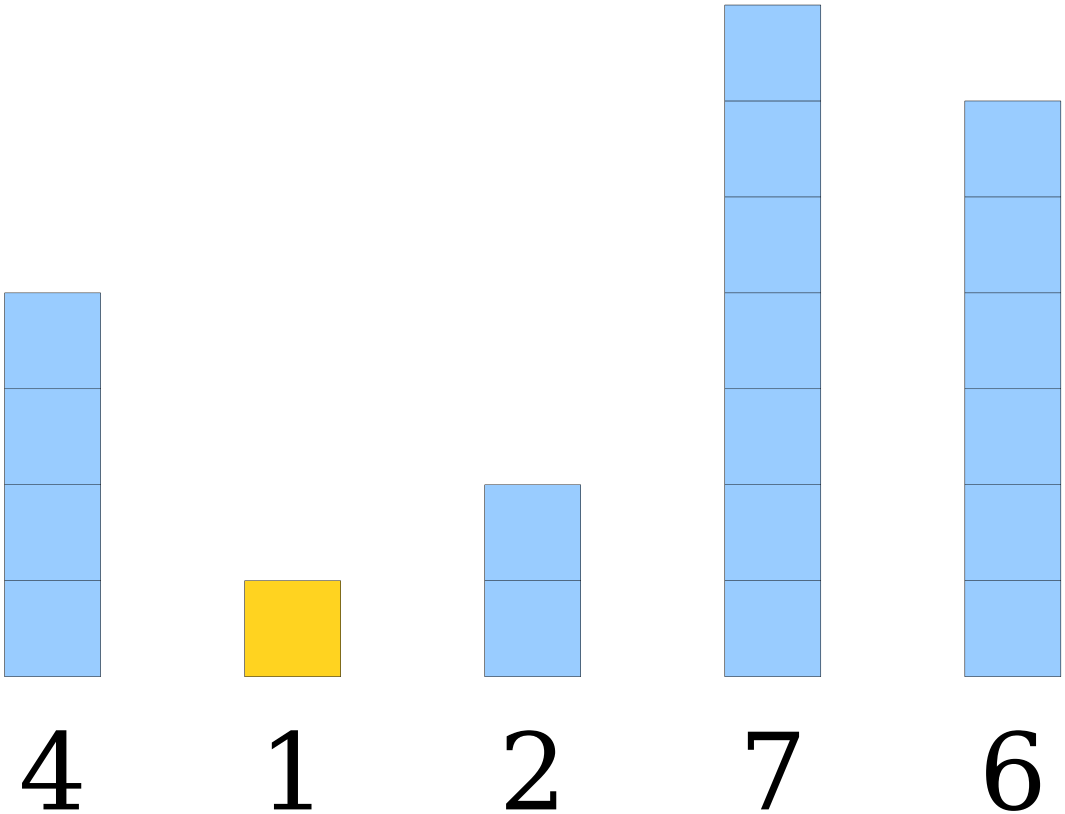
Recap from Last Time

An Initial Idea: ***Selection Sort***

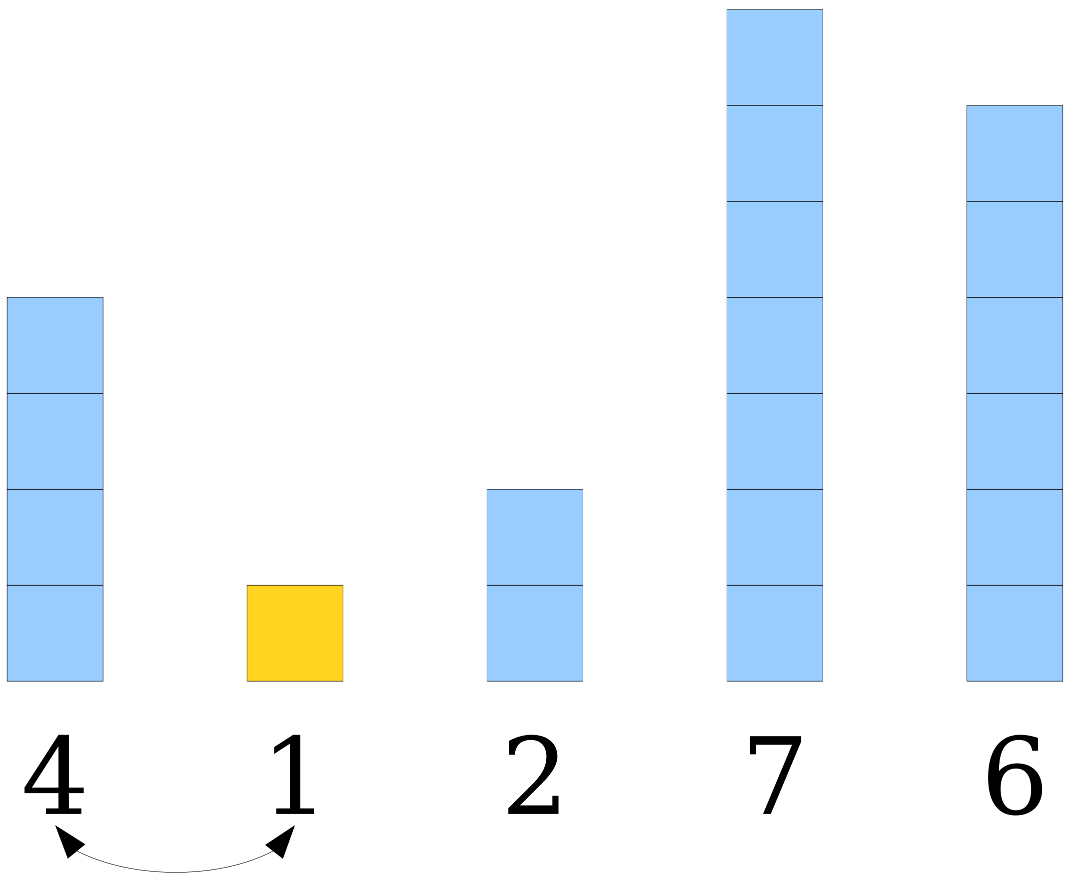
An Initial Idea: ***Selection Sort***



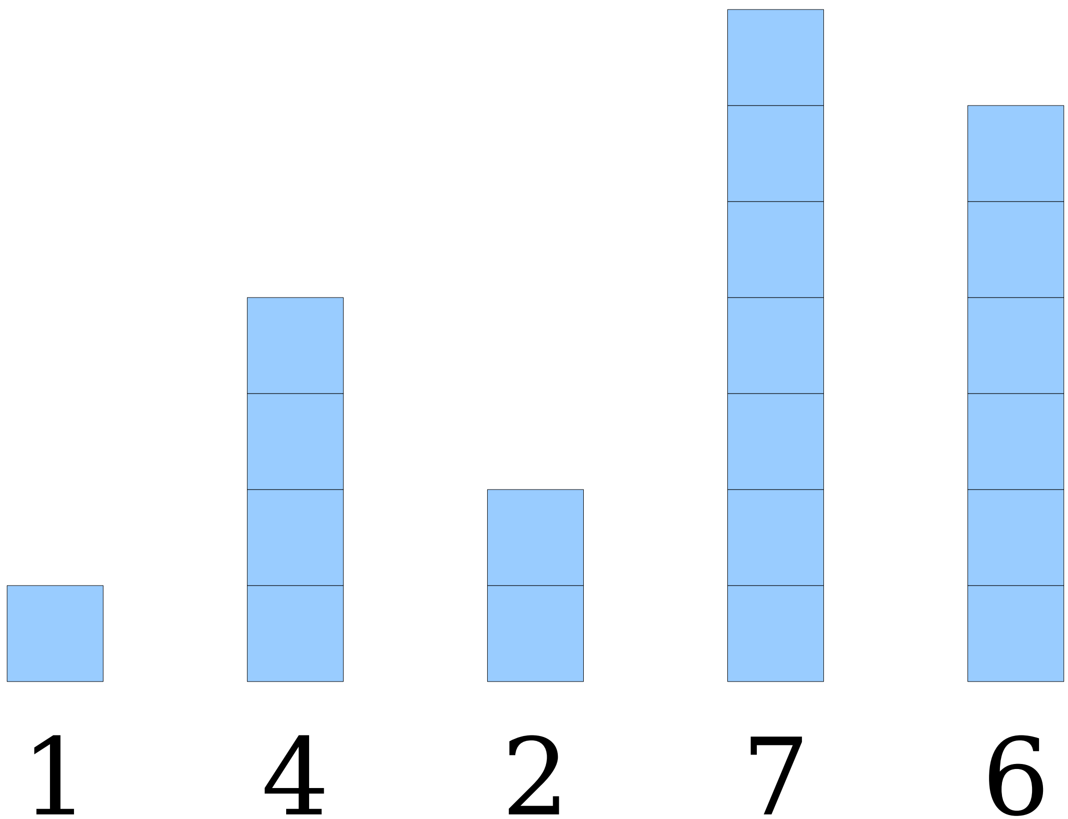
An Initial Idea: ***Selection Sort***



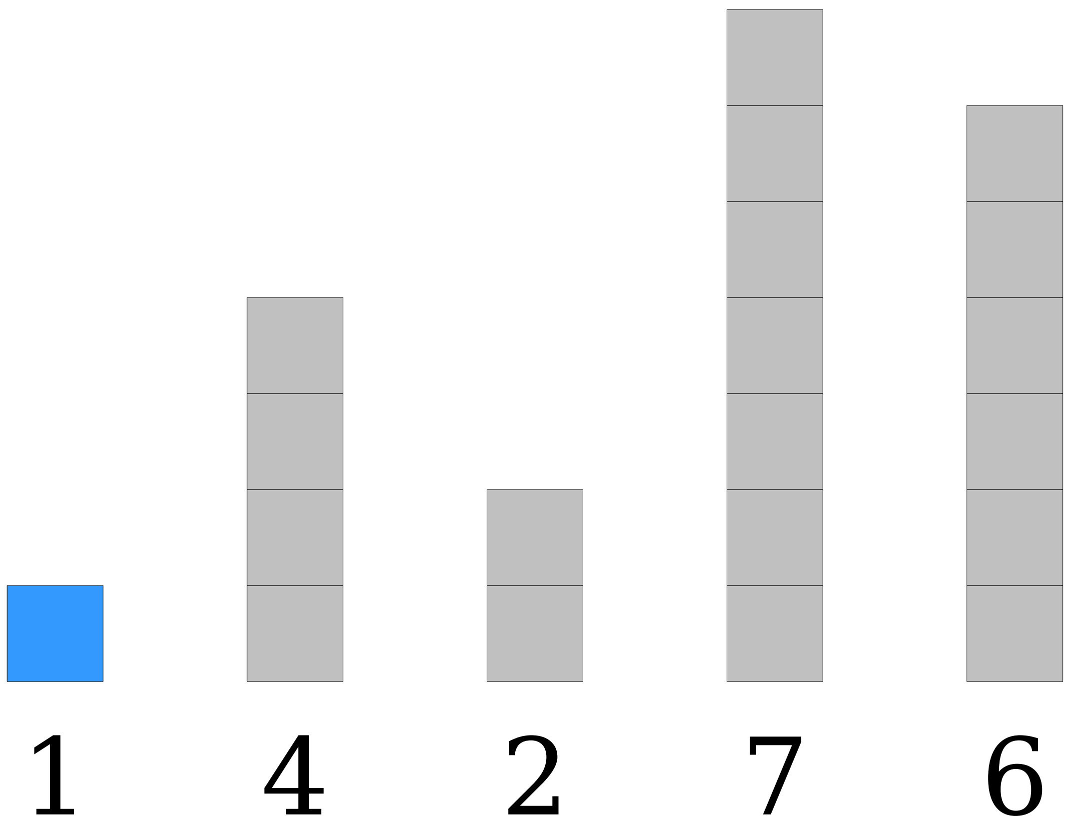
An Initial Idea: ***Selection Sort***



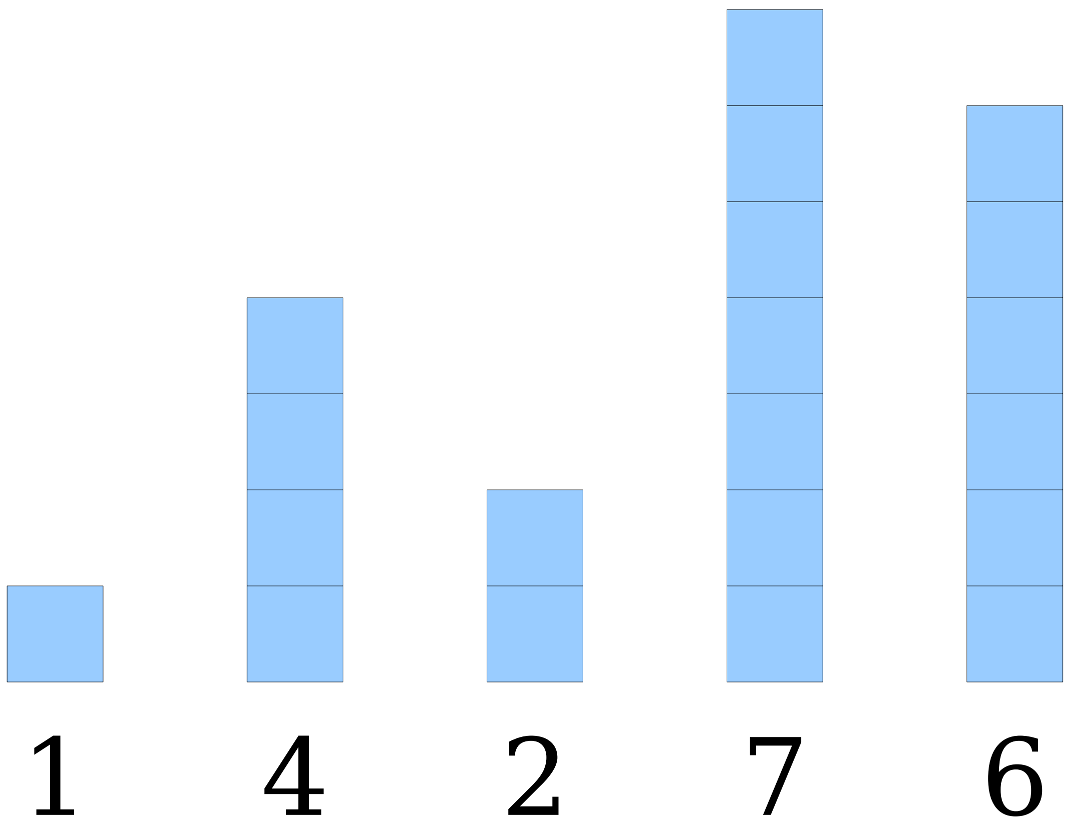
An Initial Idea: ***Selection Sort***



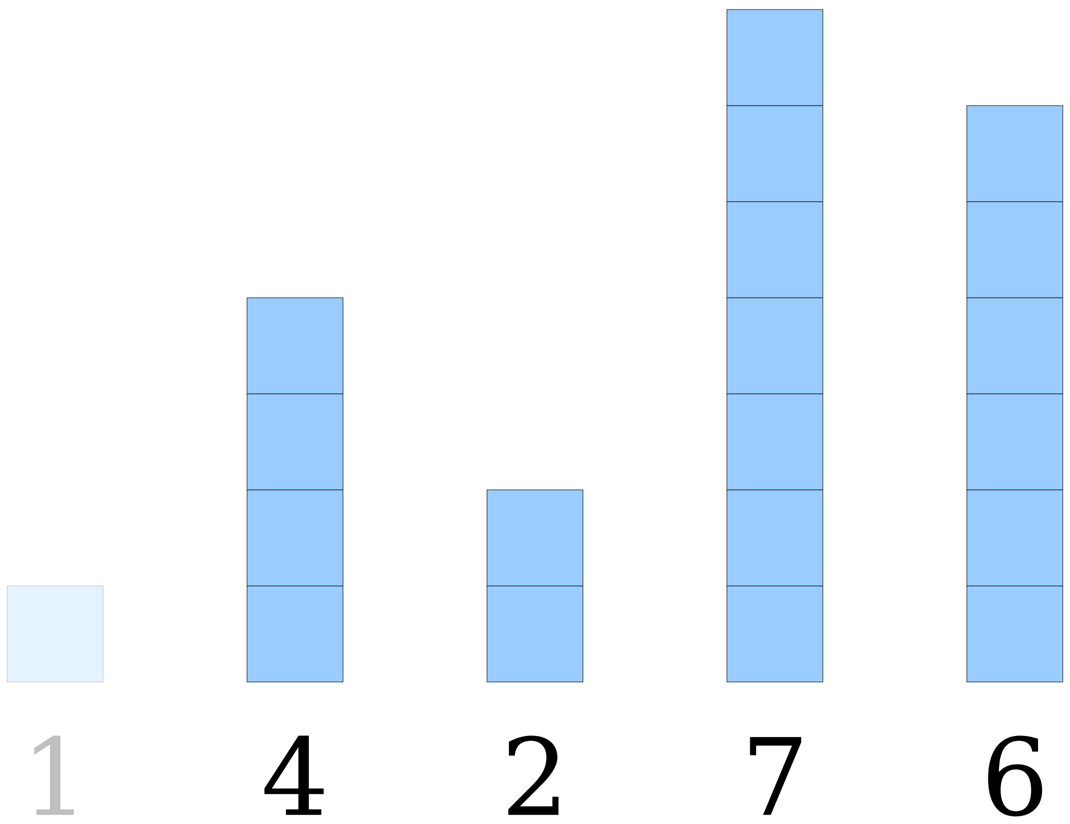
An Initial Idea: ***Selection Sort***



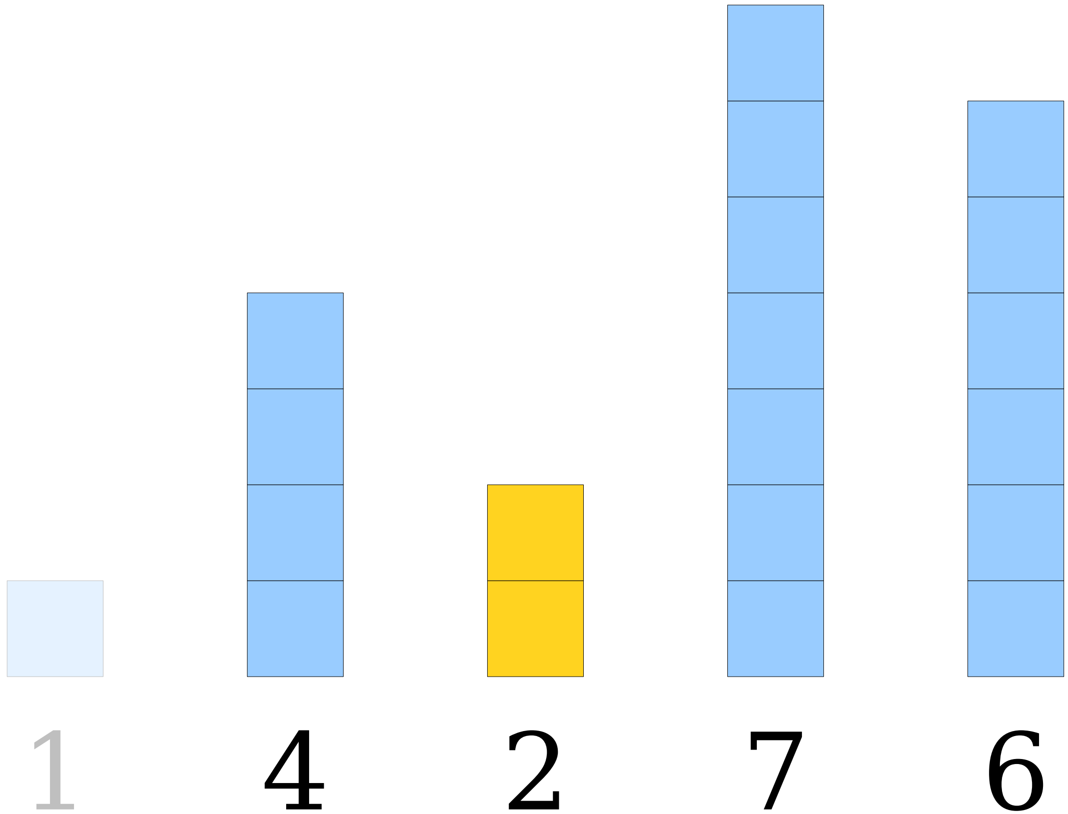
An Initial Idea: ***Selection Sort***



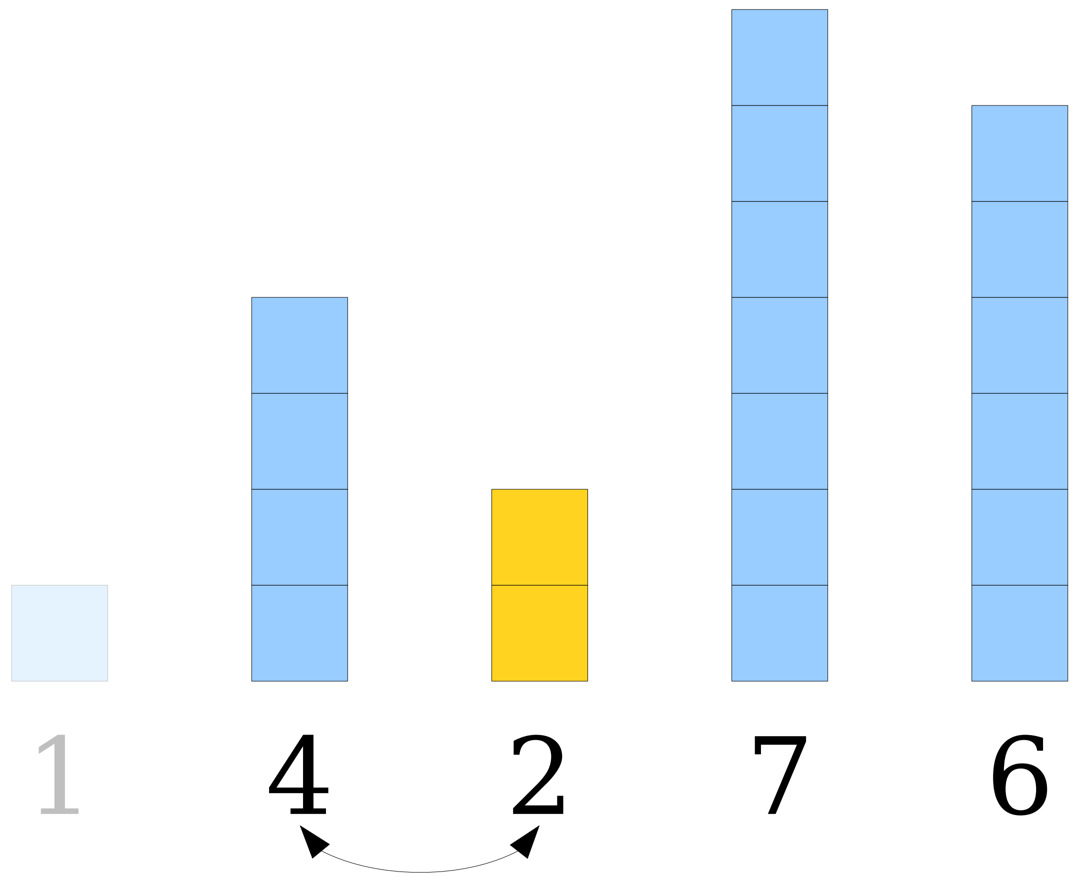
An Initial Idea: ***Selection Sort***



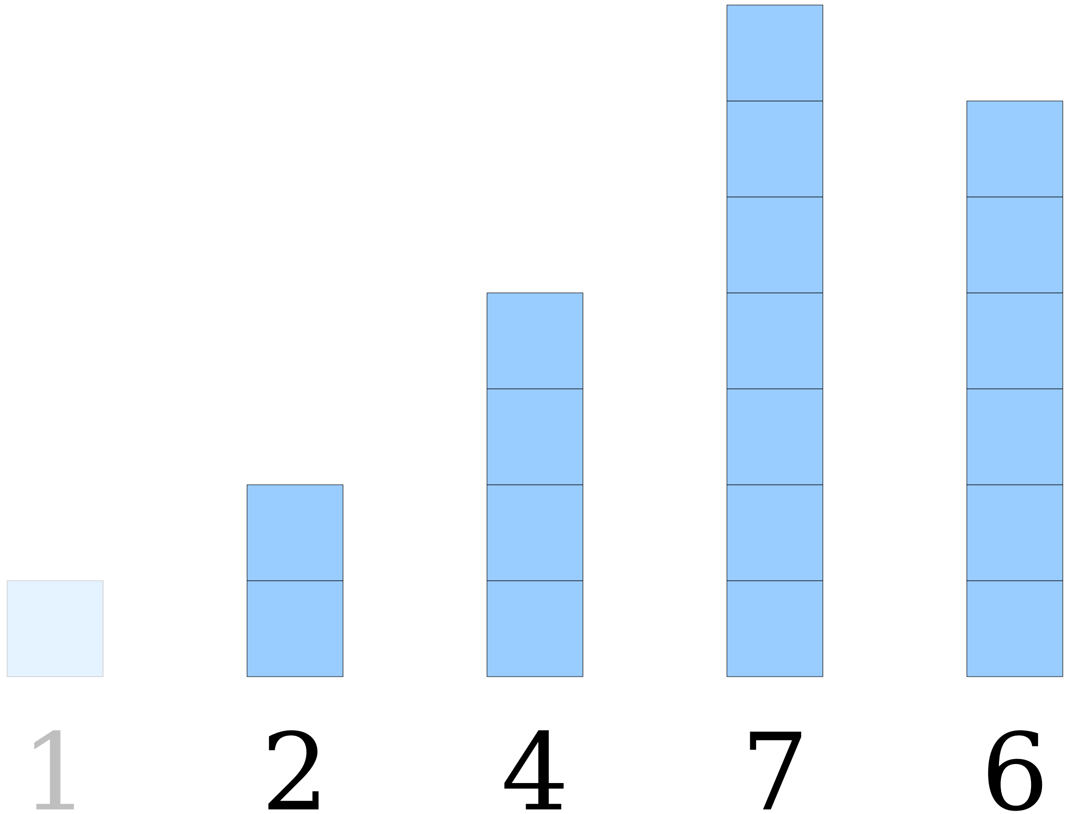
An Initial Idea: ***Selection Sort***



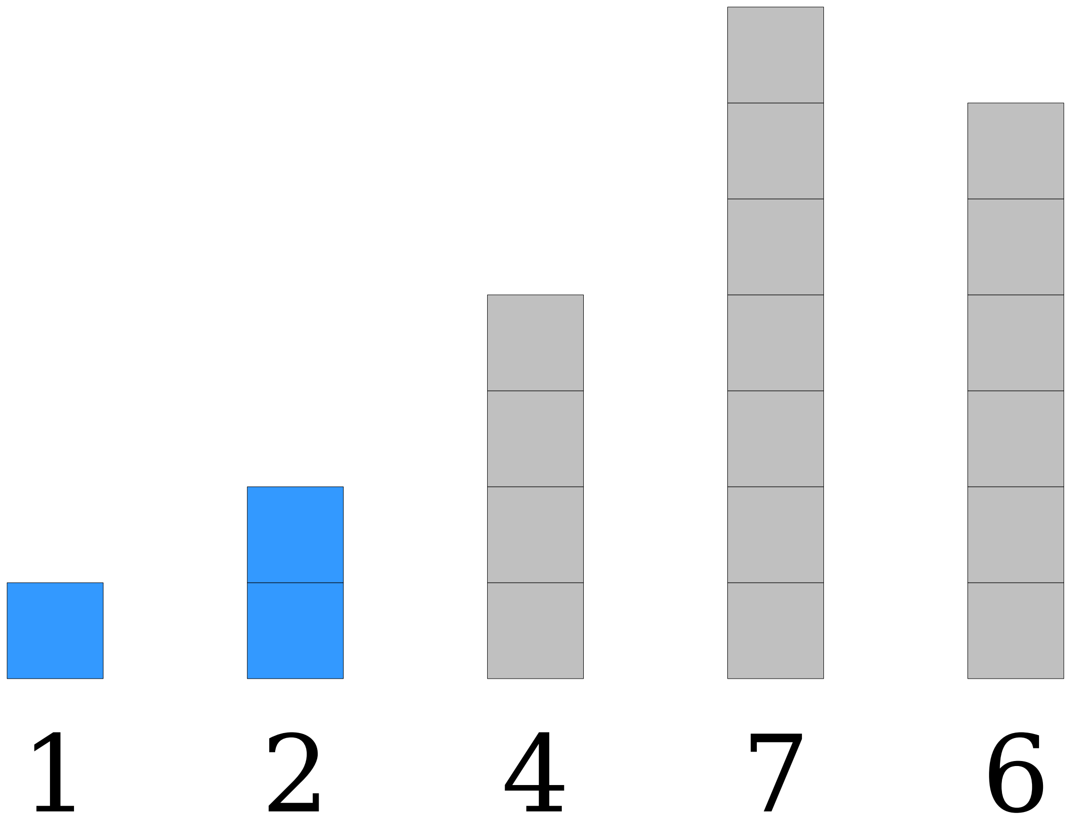
An Initial Idea: ***Selection Sort***



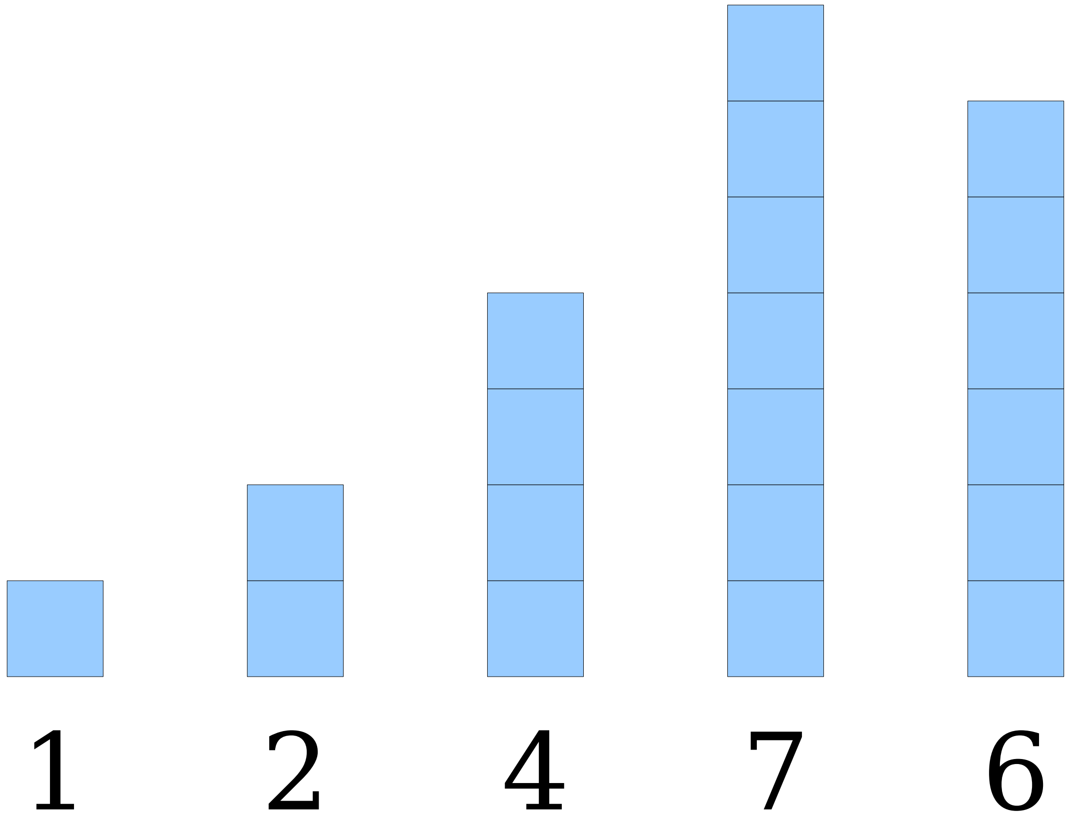
An Initial Idea: ***Selection Sort***



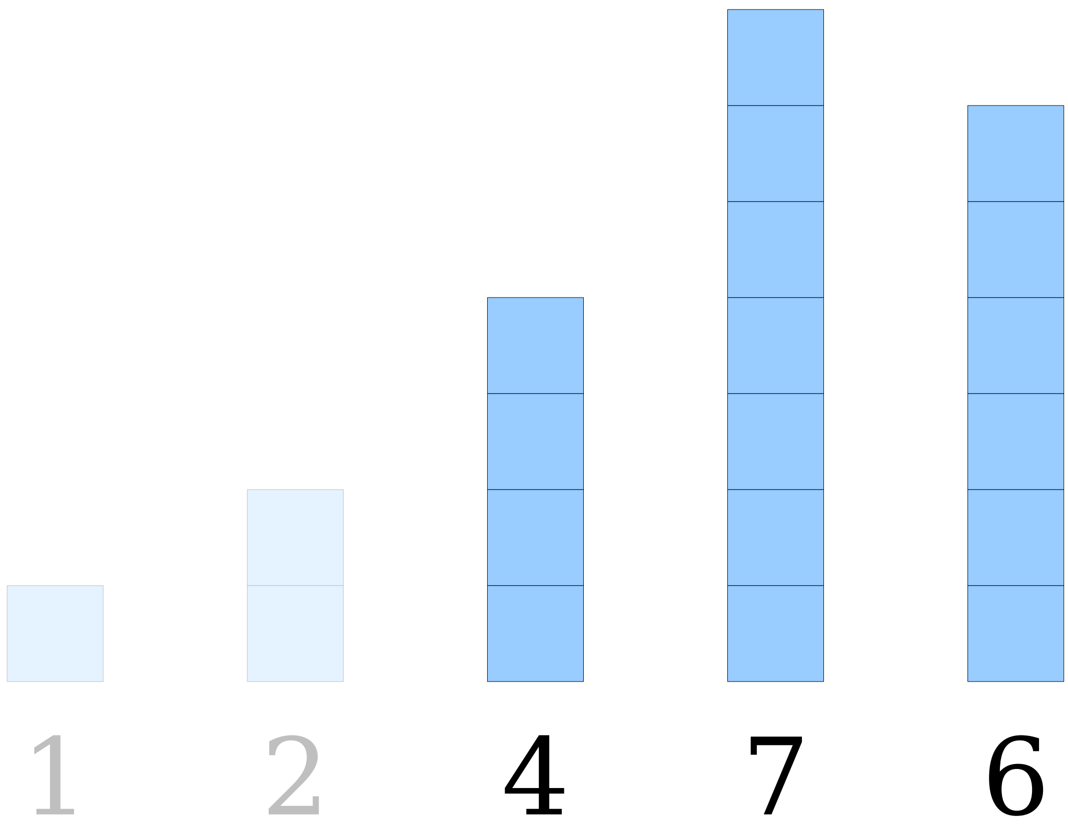
An Initial Idea: ***Selection Sort***



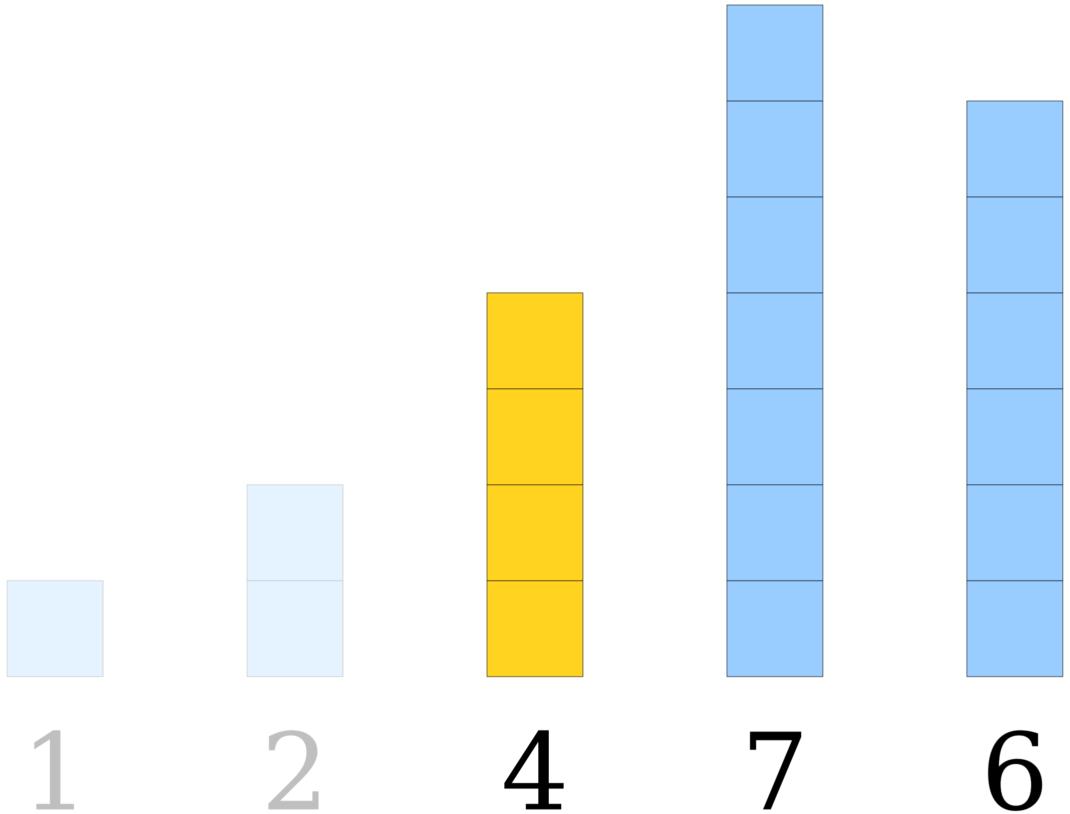
An Initial Idea: ***Selection Sort***



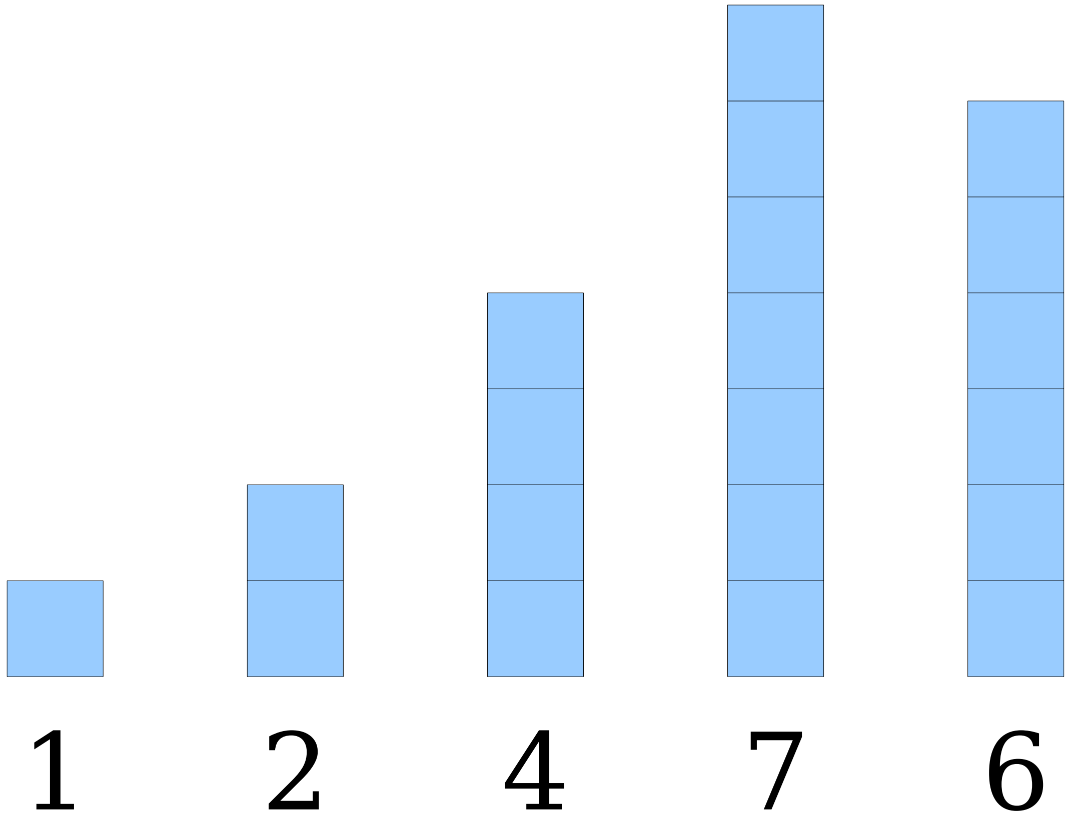
An Initial Idea: ***Selection Sort***



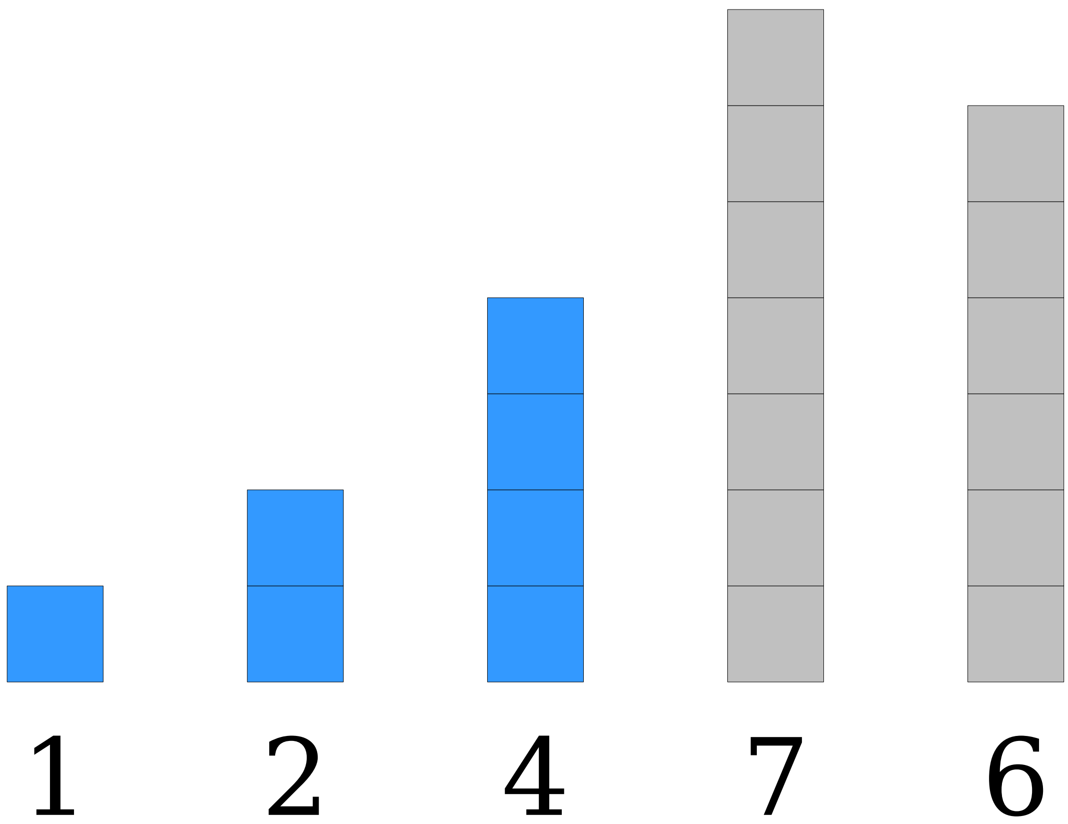
An Initial Idea: ***Selection Sort***



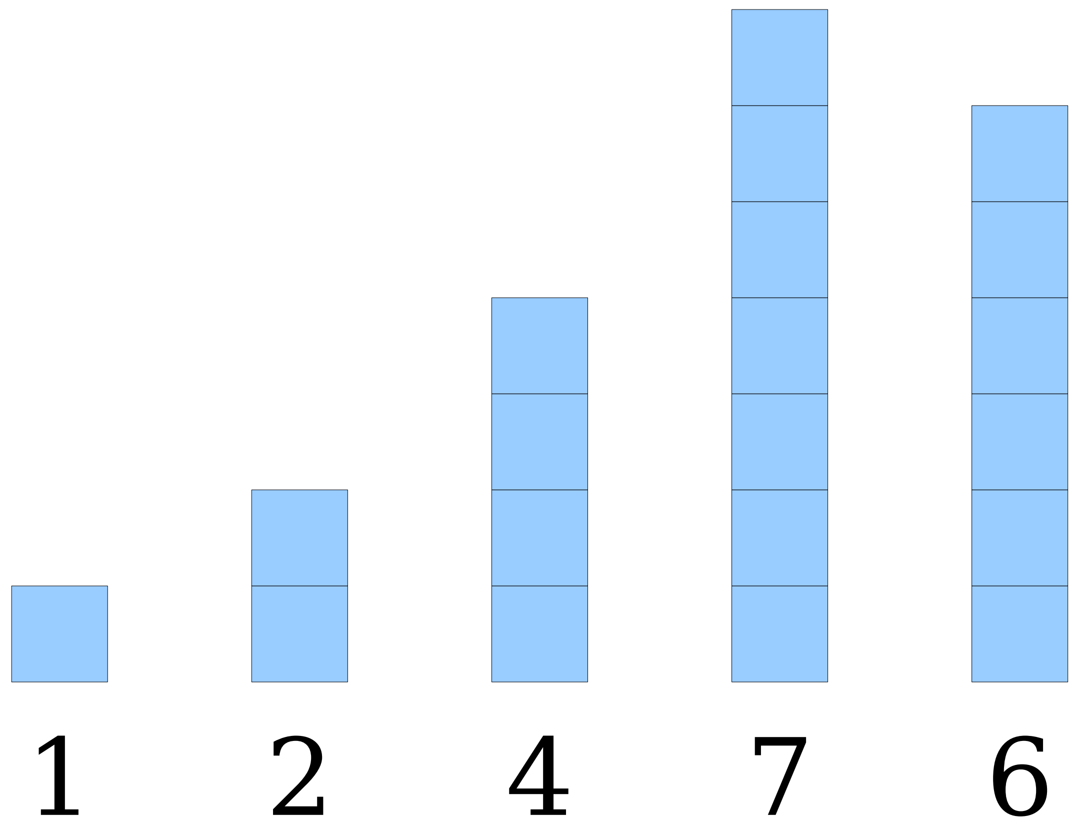
An Initial Idea: ***Selection Sort***



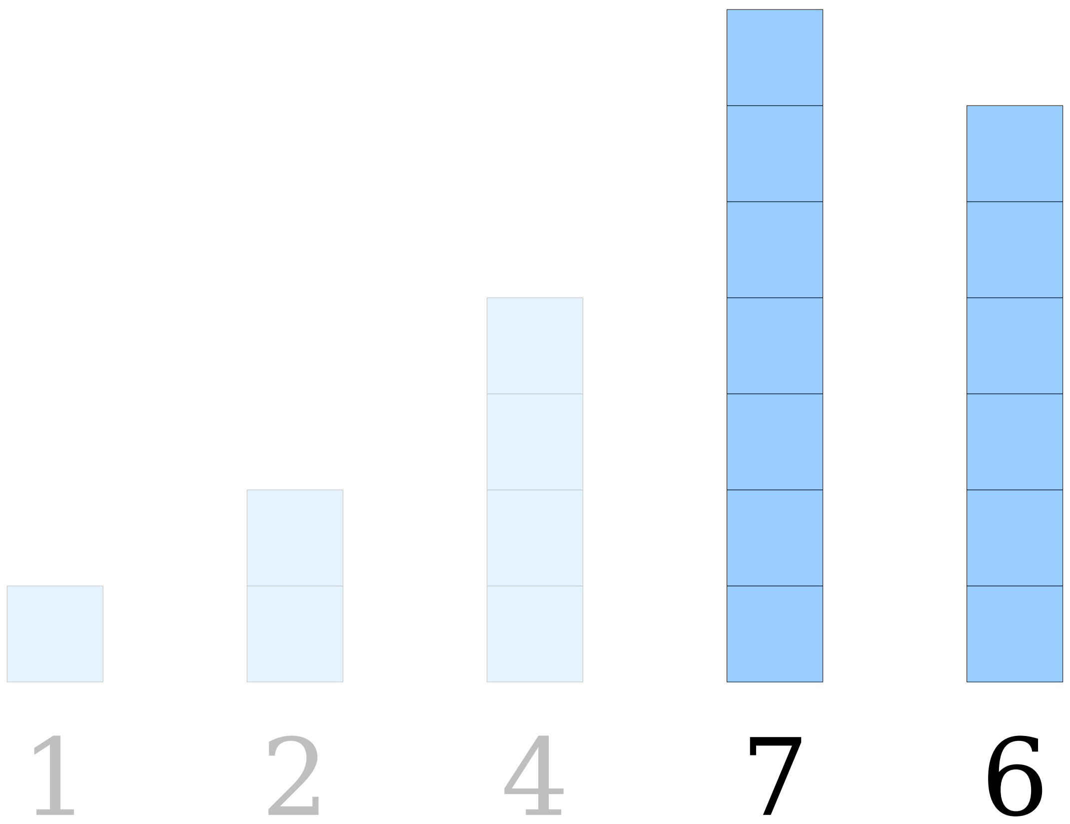
An Initial Idea: ***Selection Sort***



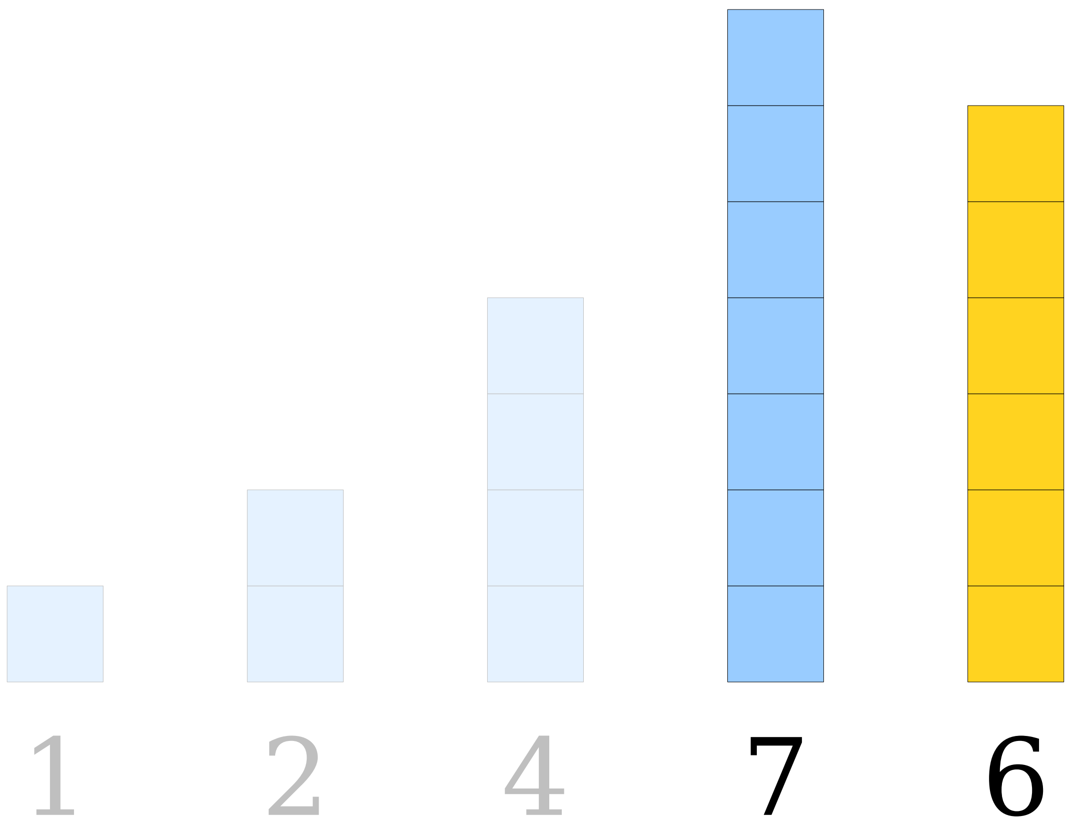
An Initial Idea: ***Selection Sort***



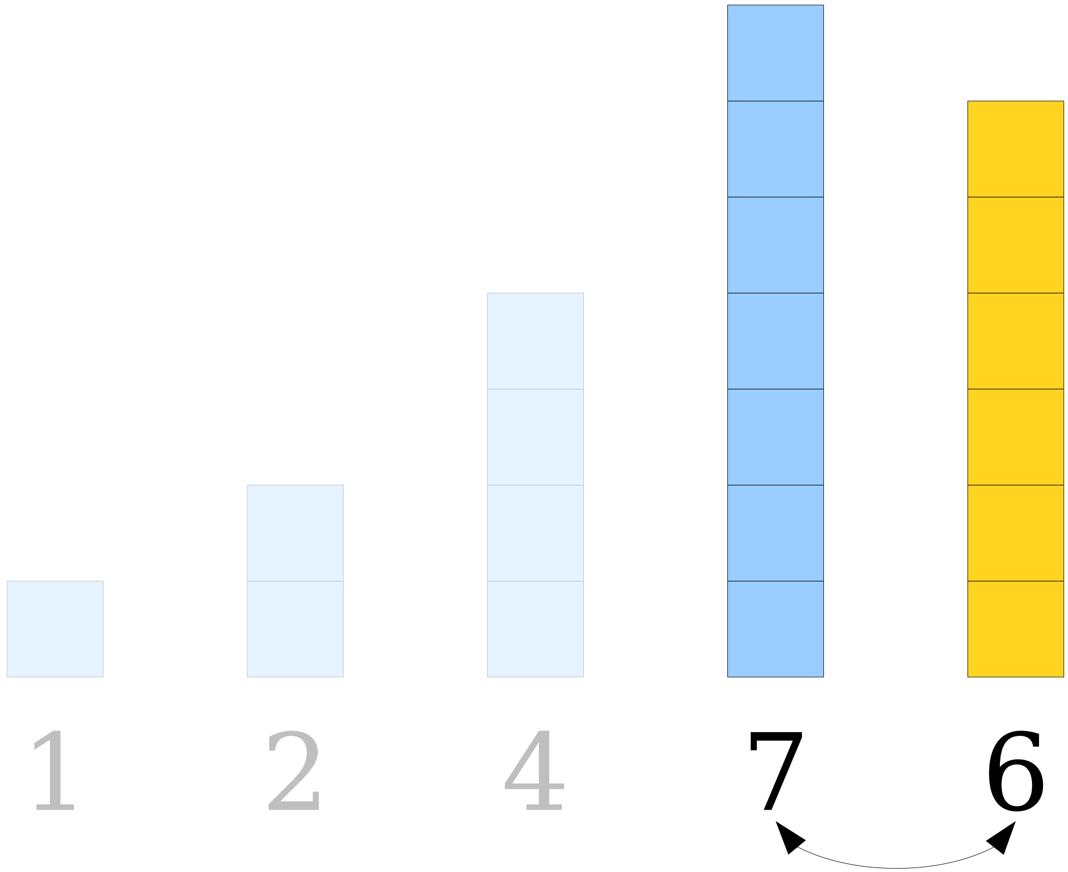
An Initial Idea: ***Selection Sort***



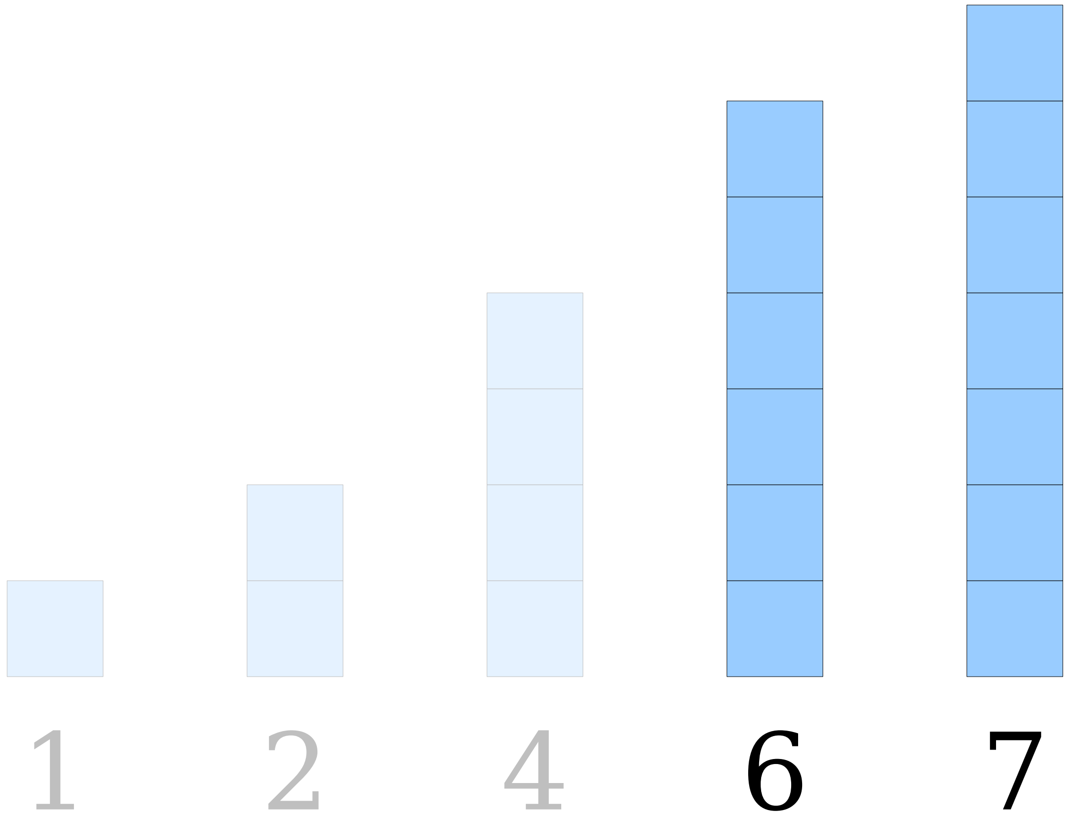
An Initial Idea: ***Selection Sort***



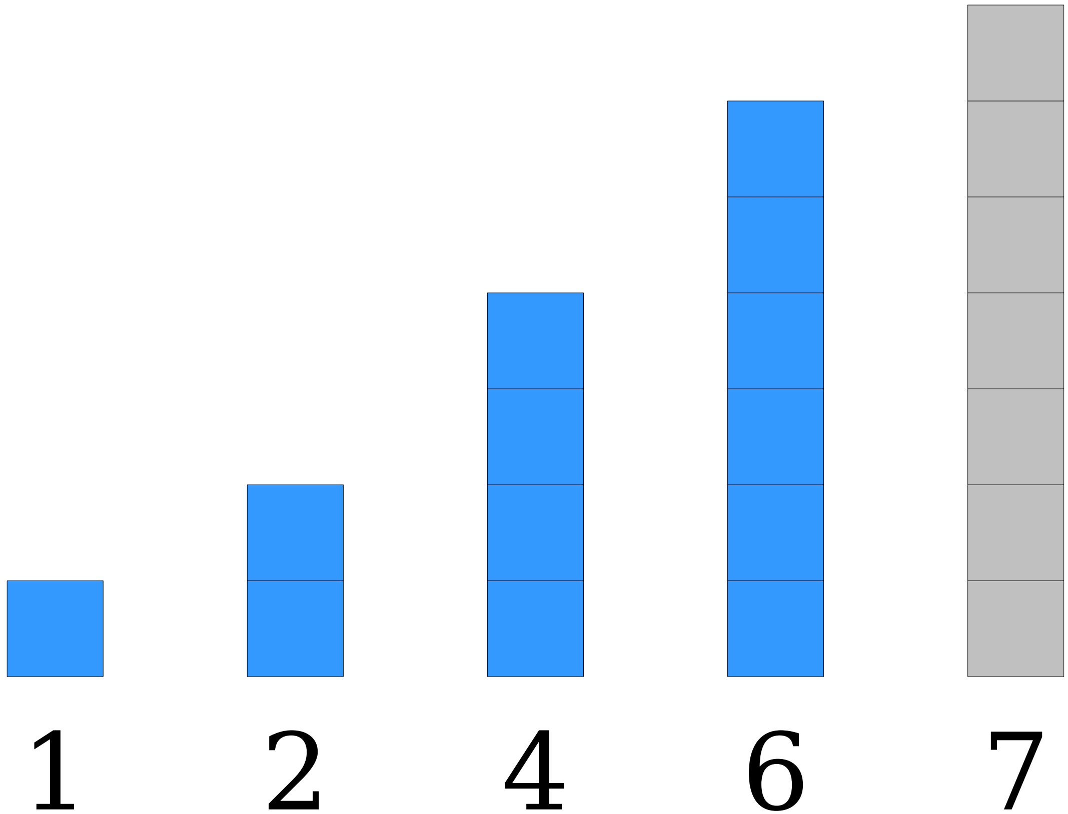
An Initial Idea: ***Selection Sort***



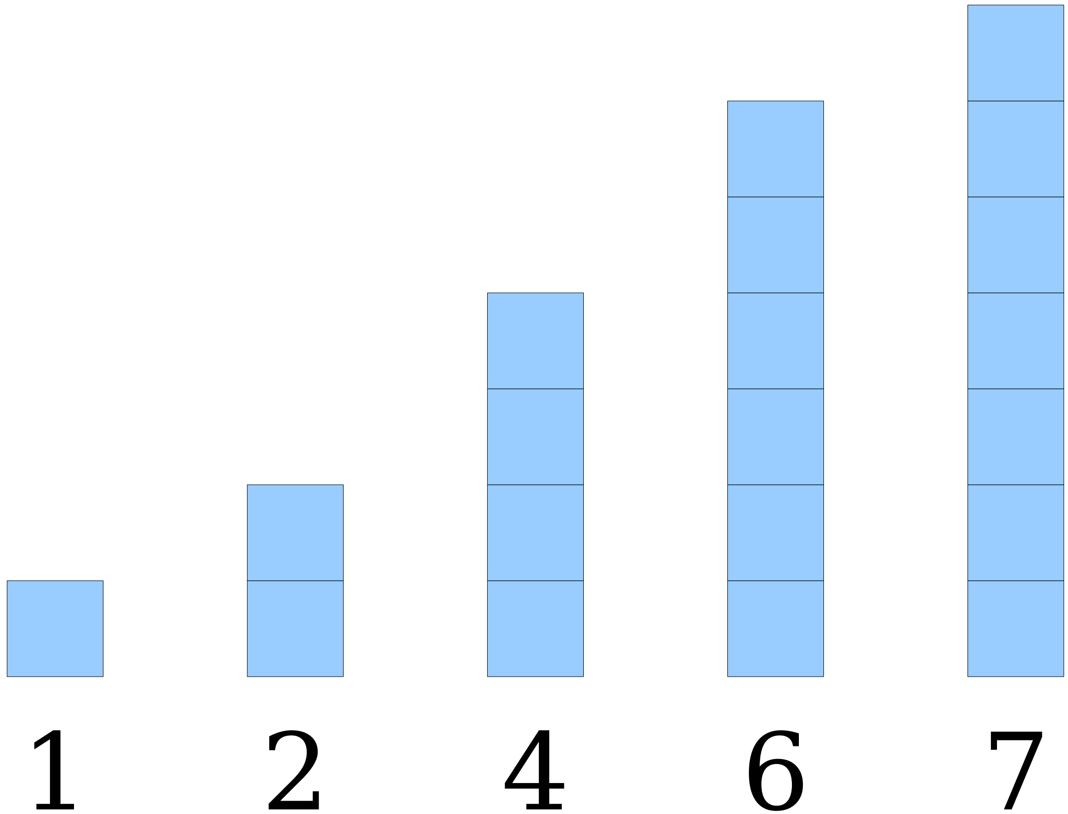
An Initial Idea: ***Selection Sort***



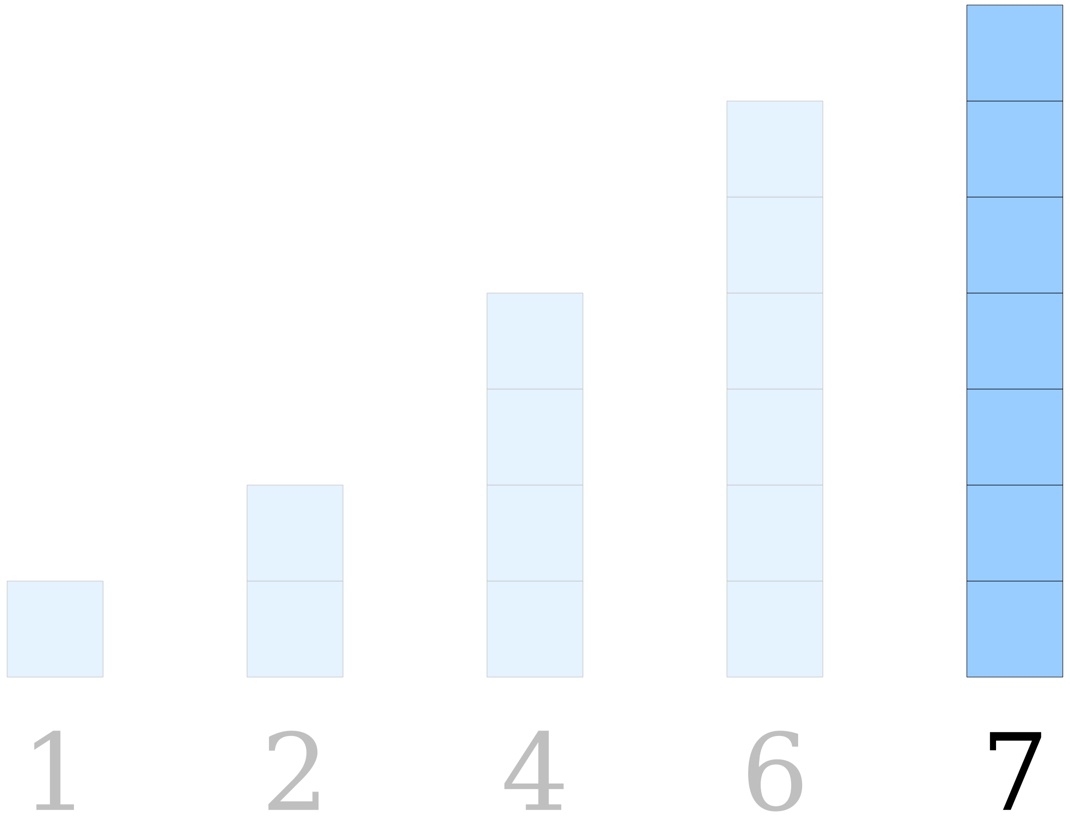
An Initial Idea: ***Selection Sort***



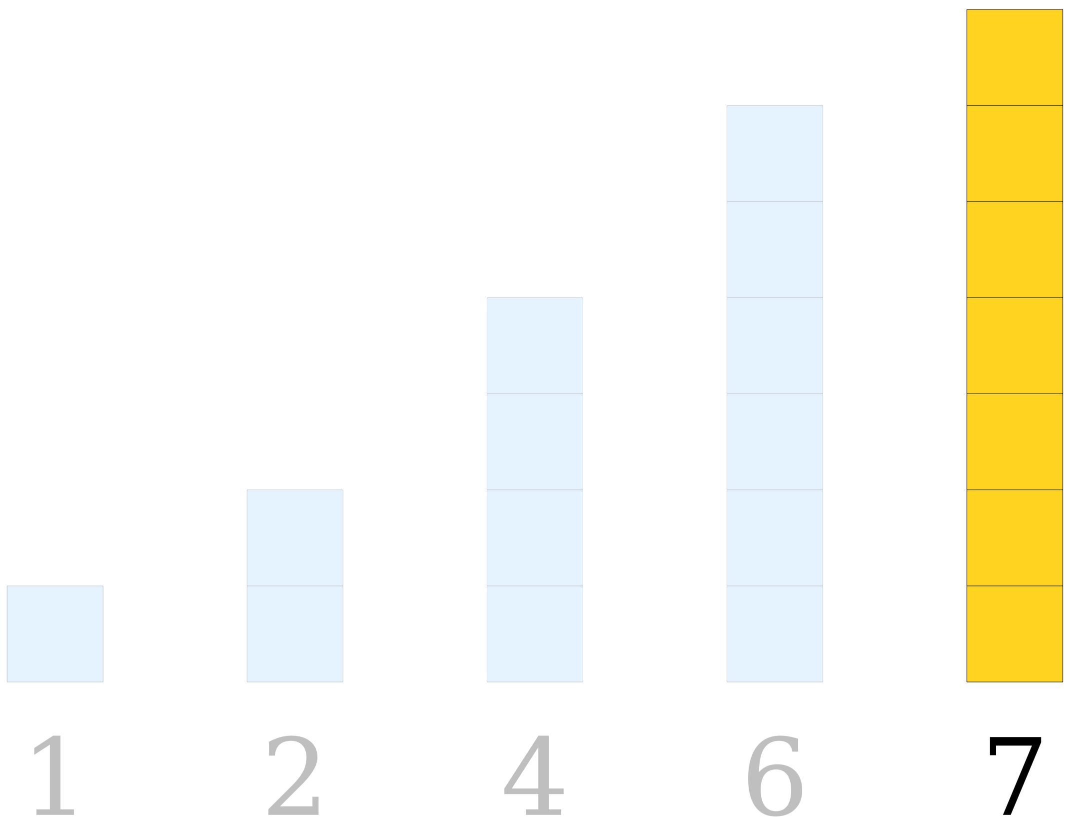
An Initial Idea: ***Selection Sort***



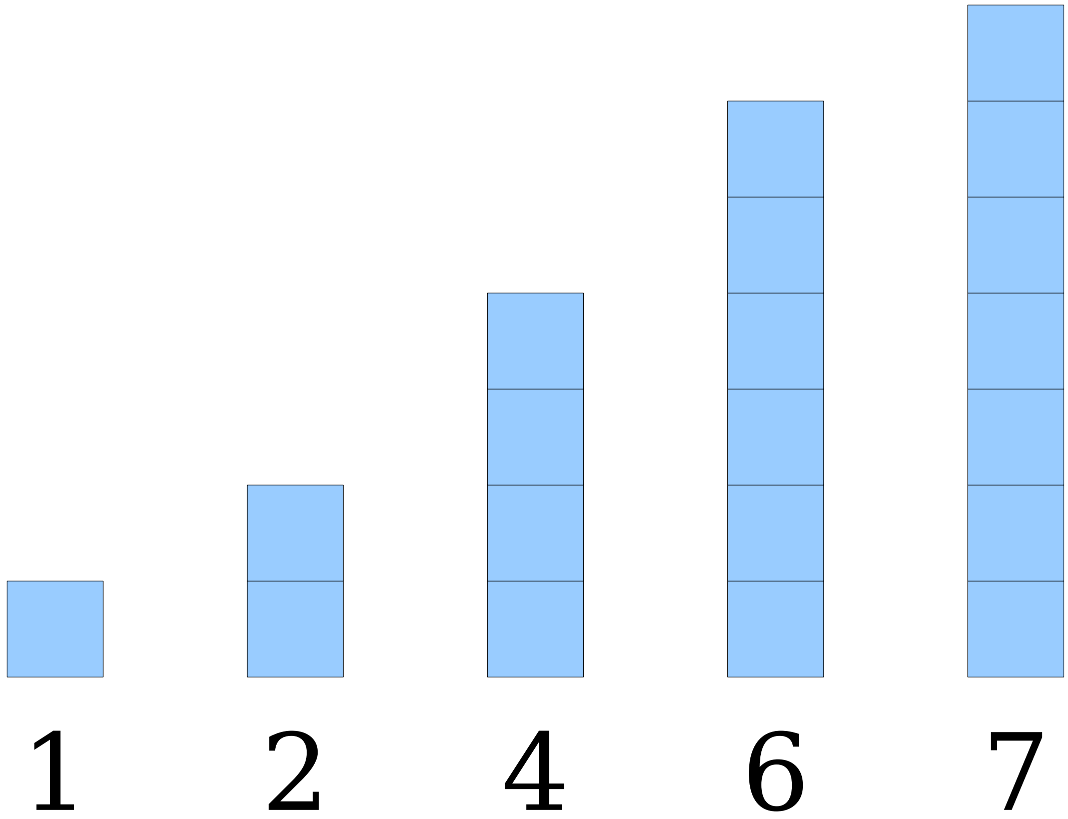
An Initial Idea: ***Selection Sort***



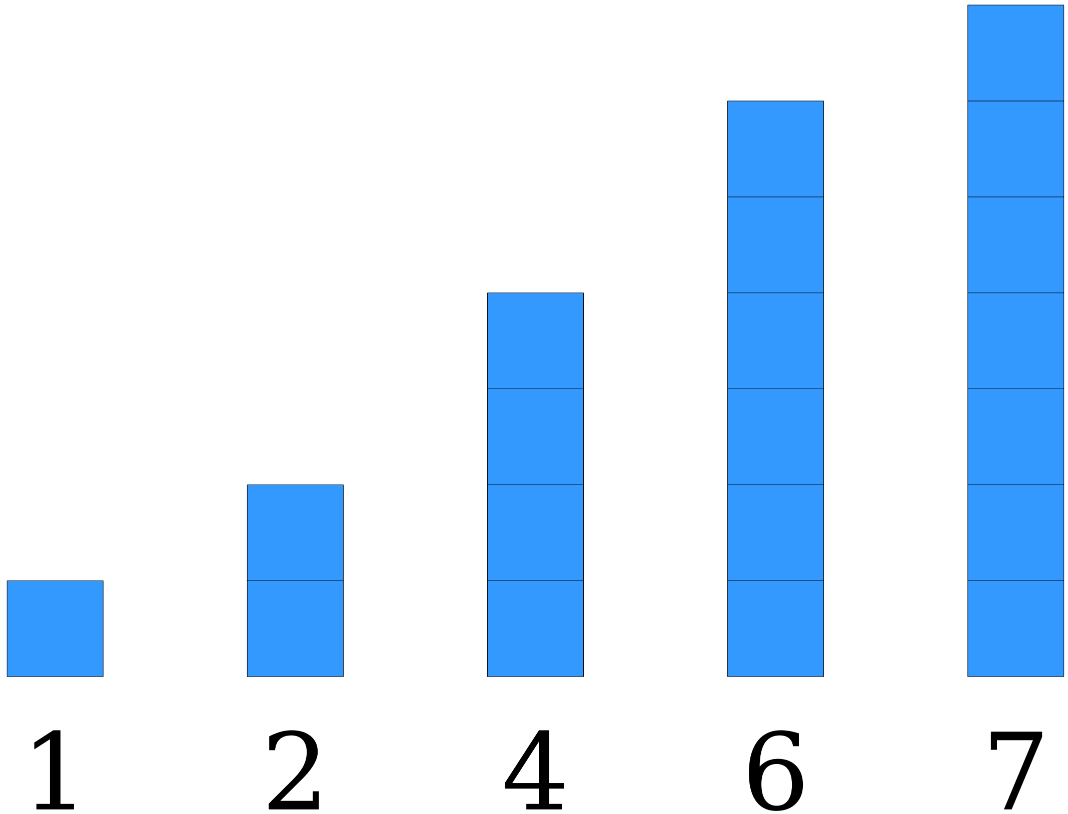
An Initial Idea: ***Selection Sort***



An Initial Idea: ***Selection Sort***

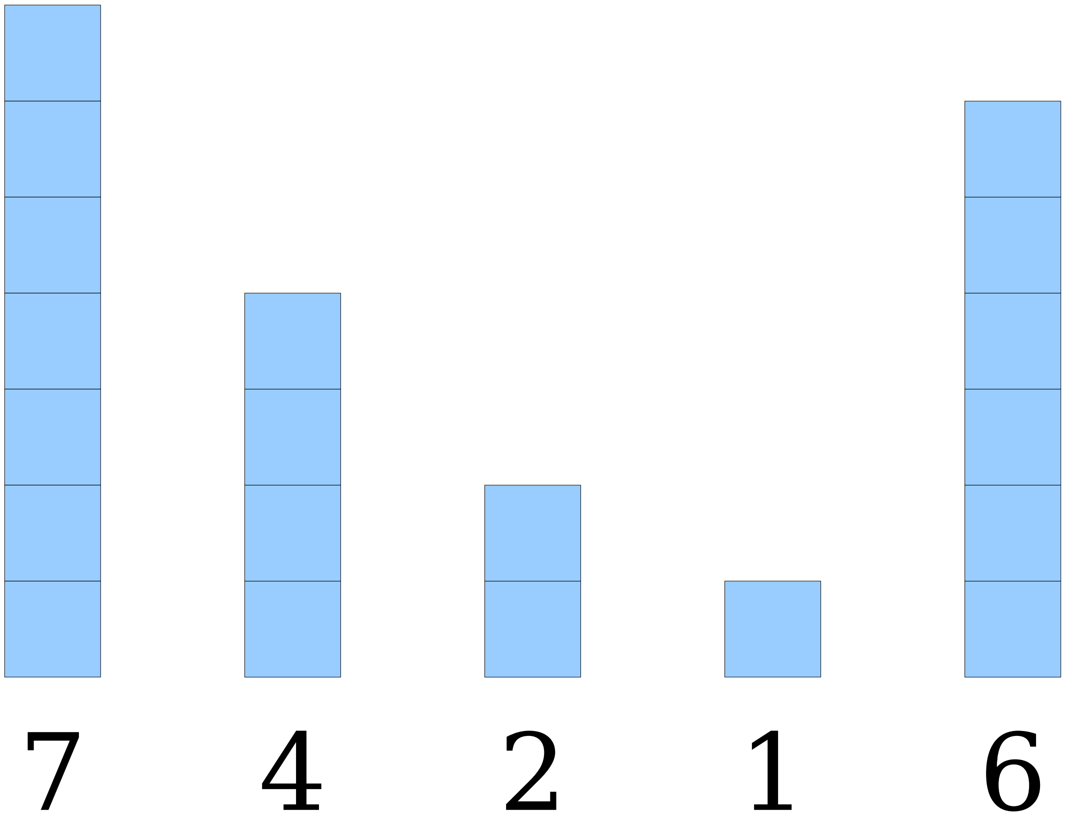


An Initial Idea: ***Selection Sort***

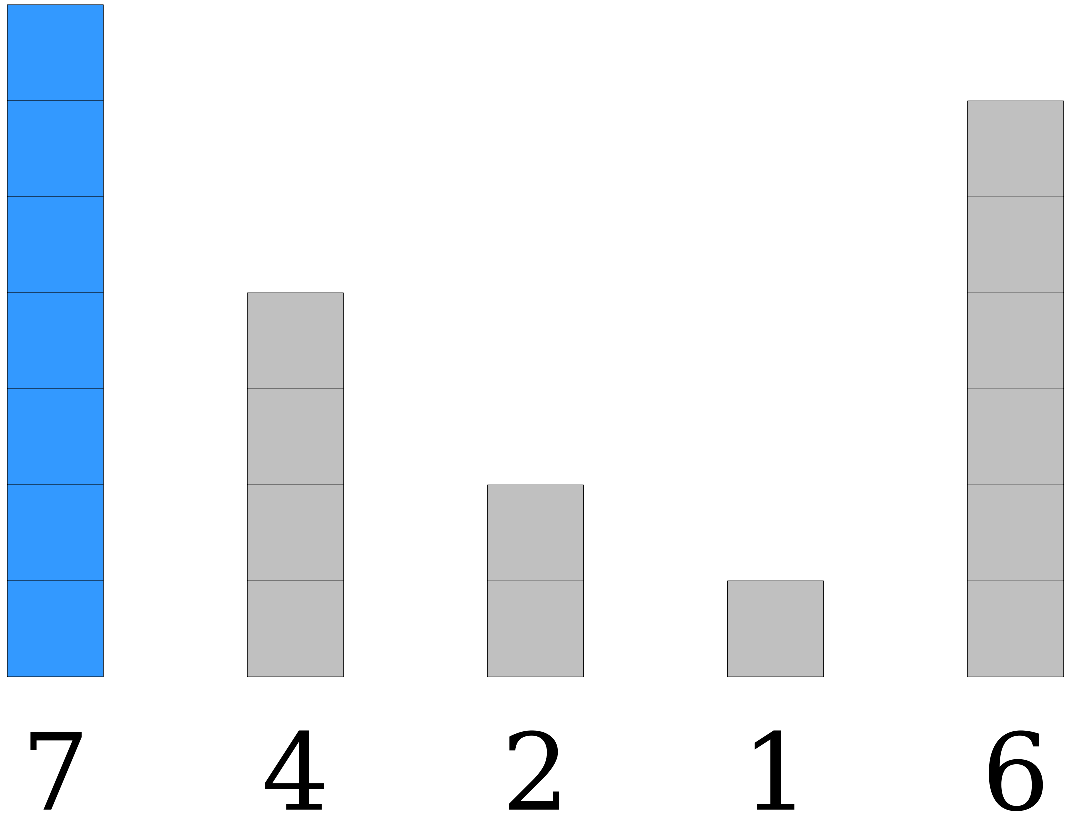


Our Next Idea: ***Insertion Sort***

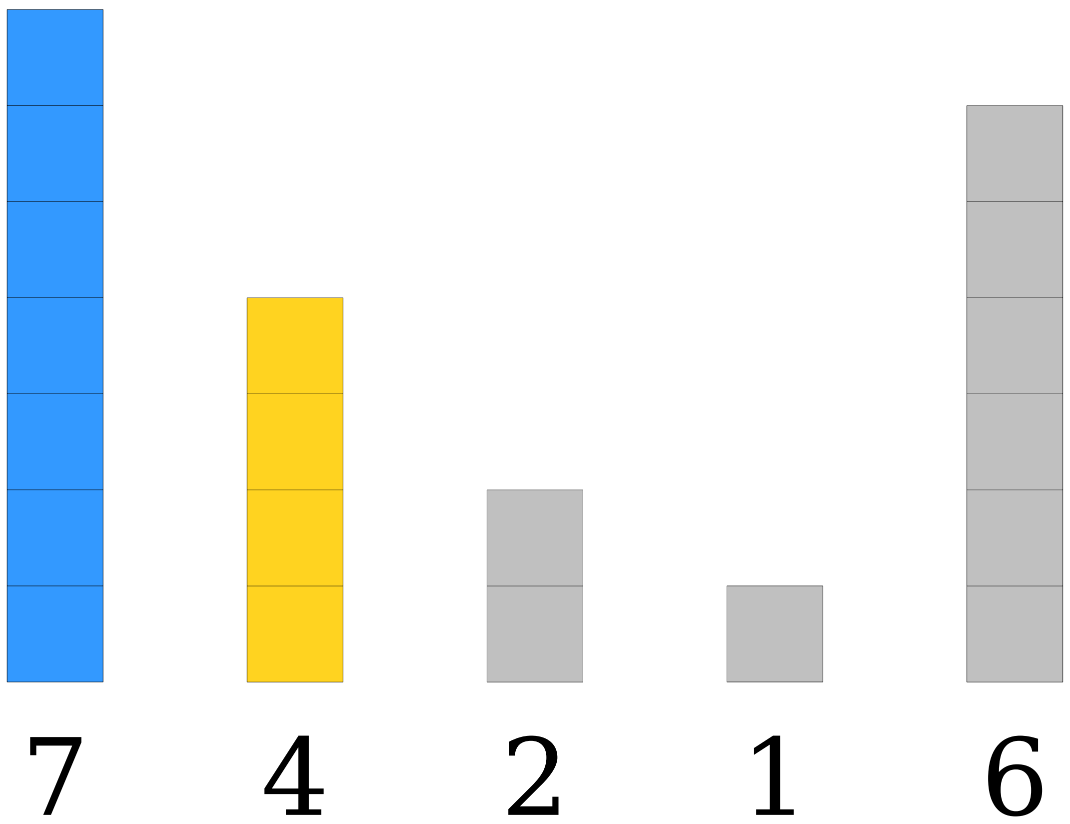
Our Next Idea: ***Insertion Sort***



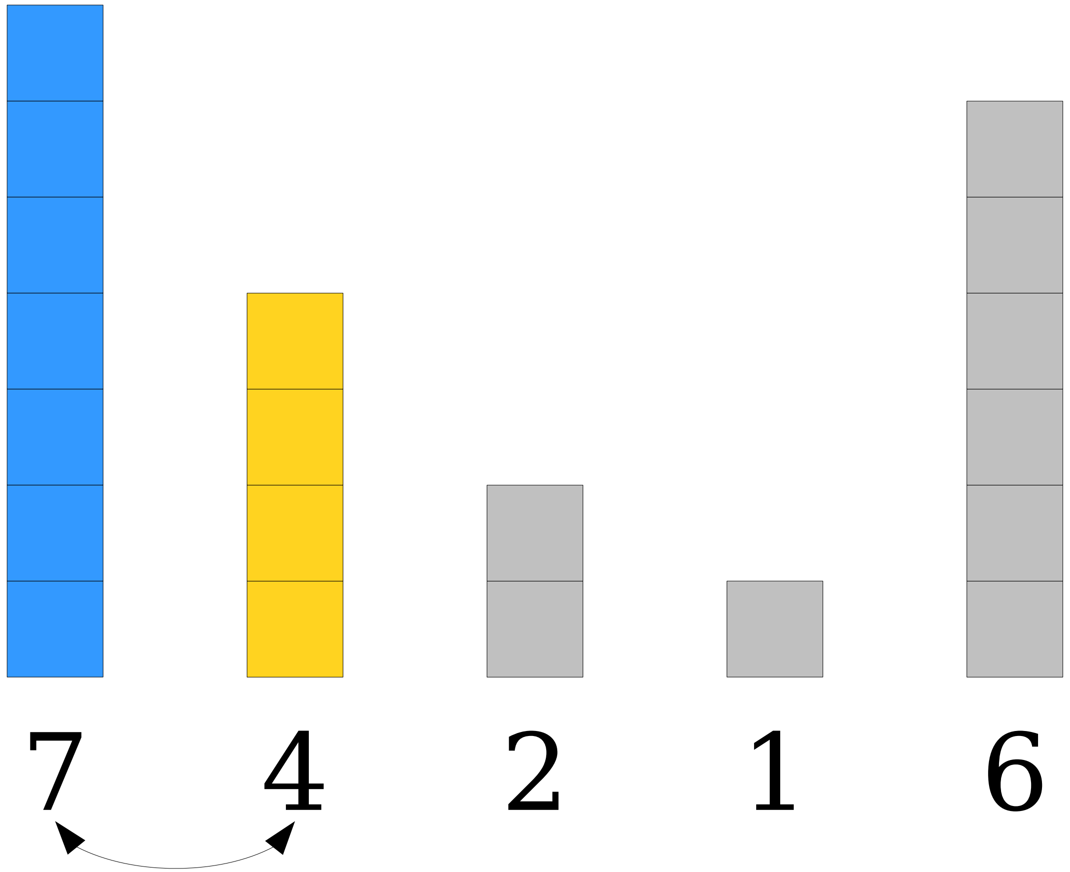
Our Next Idea: ***Insertion Sort***



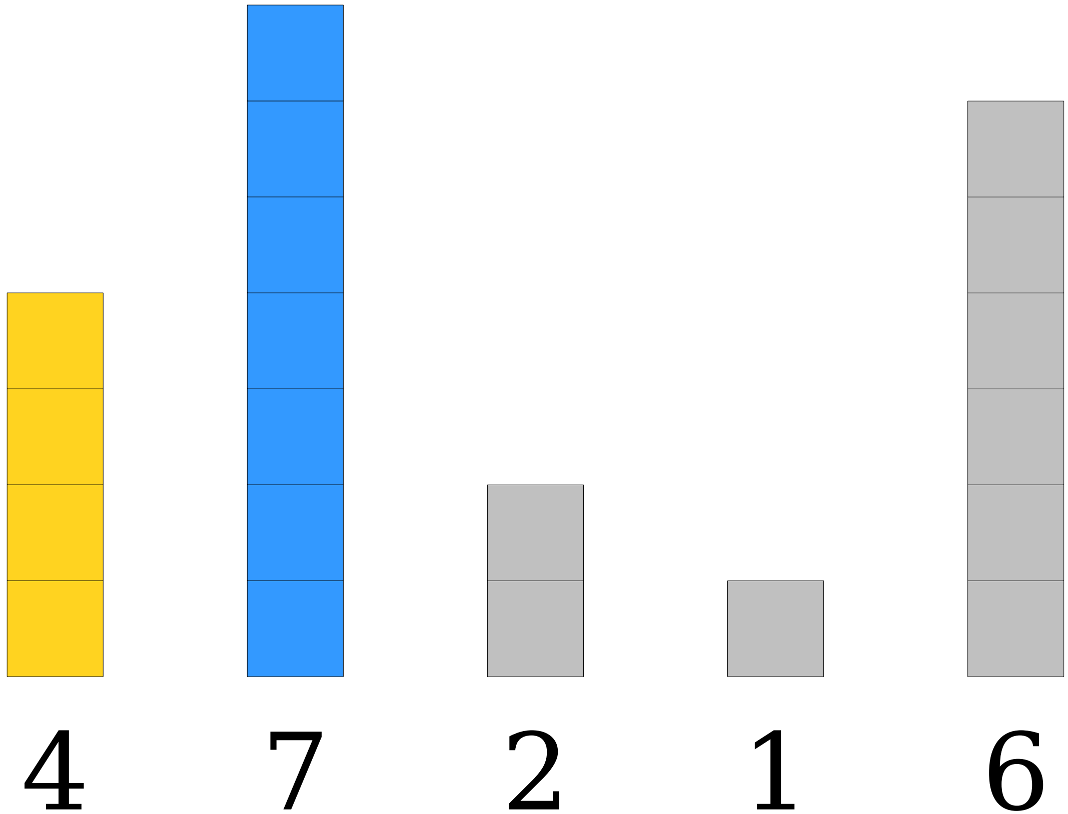
Our Next Idea: ***Insertion Sort***



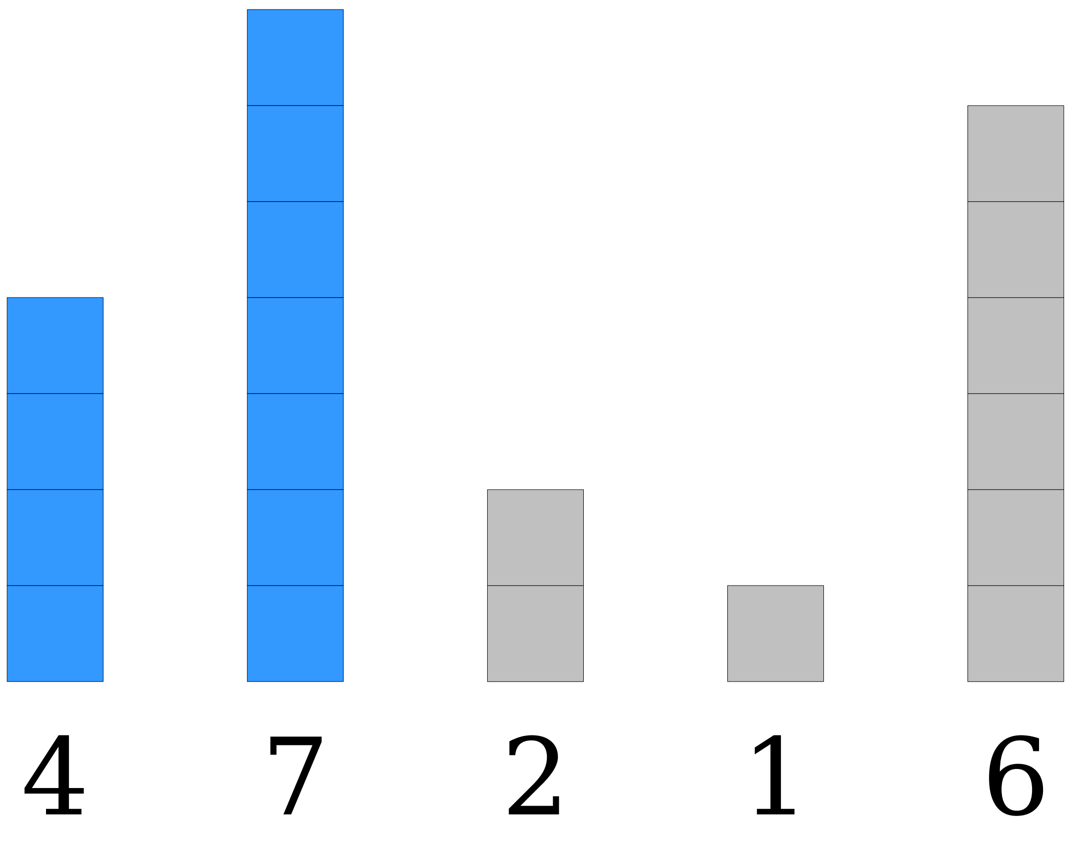
Our Next Idea: ***Insertion Sort***



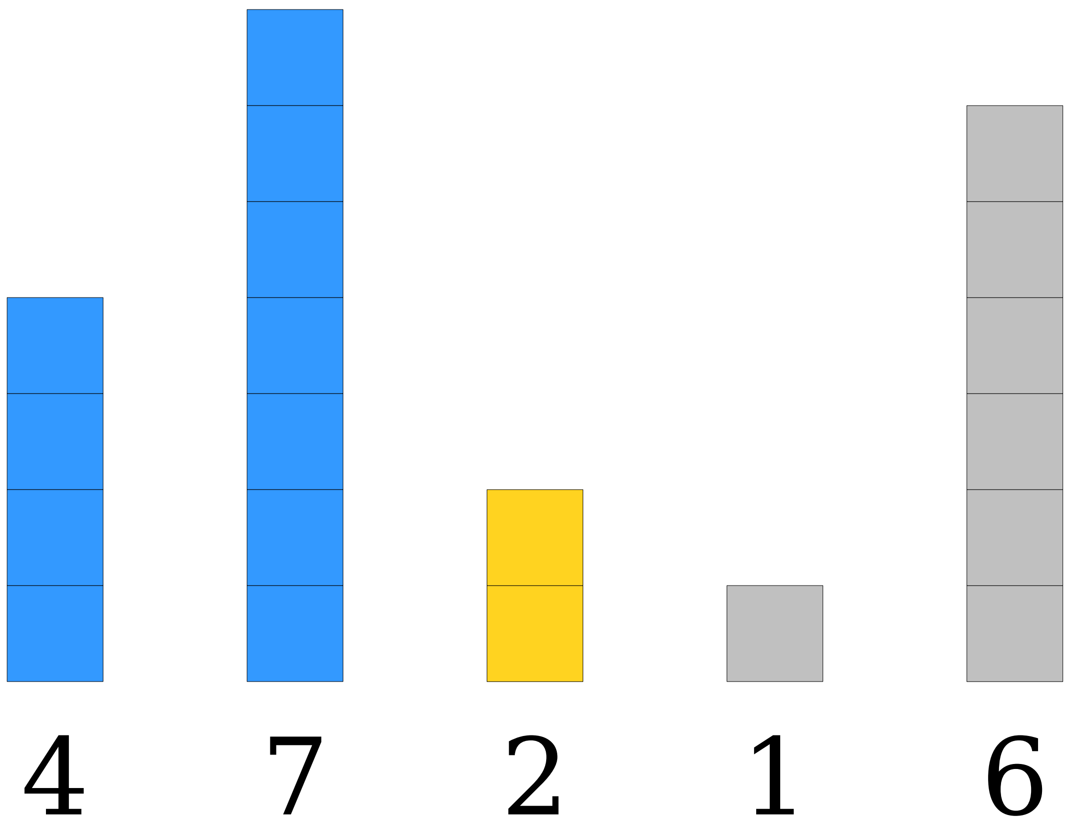
Our Next Idea: ***Insertion Sort***



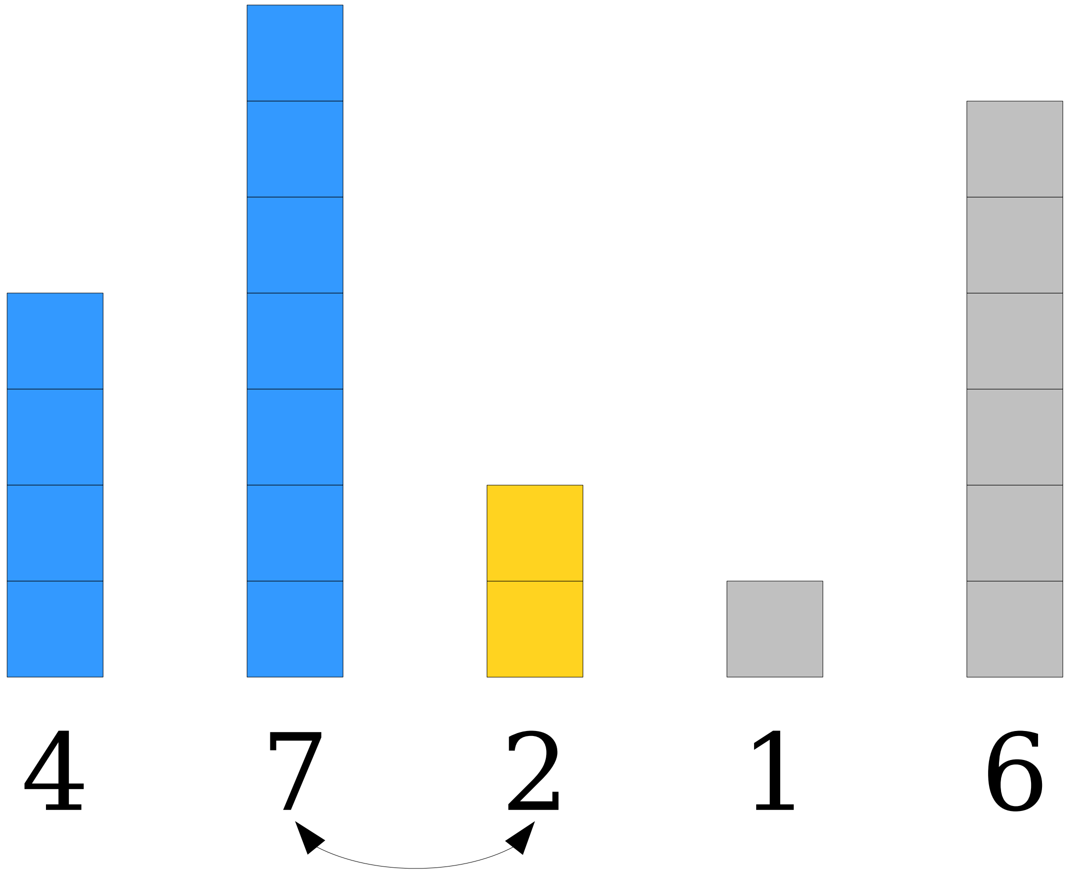
Our Next Idea: ***Insertion Sort***



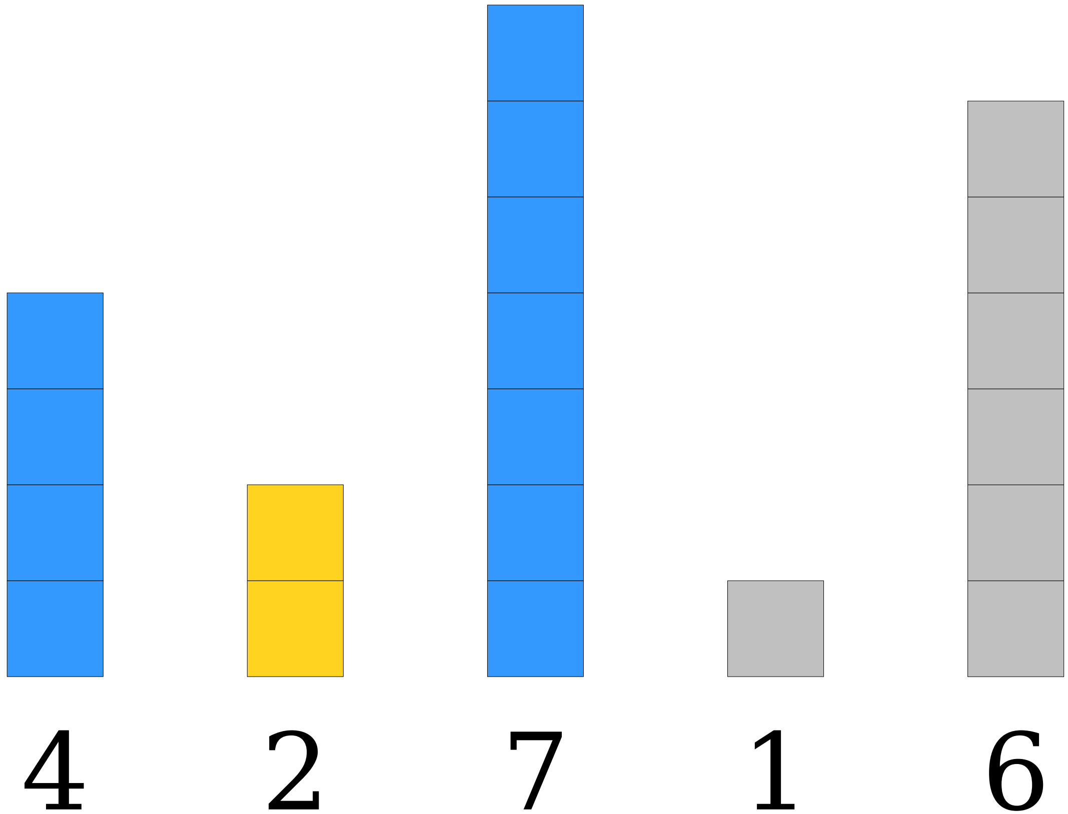
Our Next Idea: ***Insertion Sort***



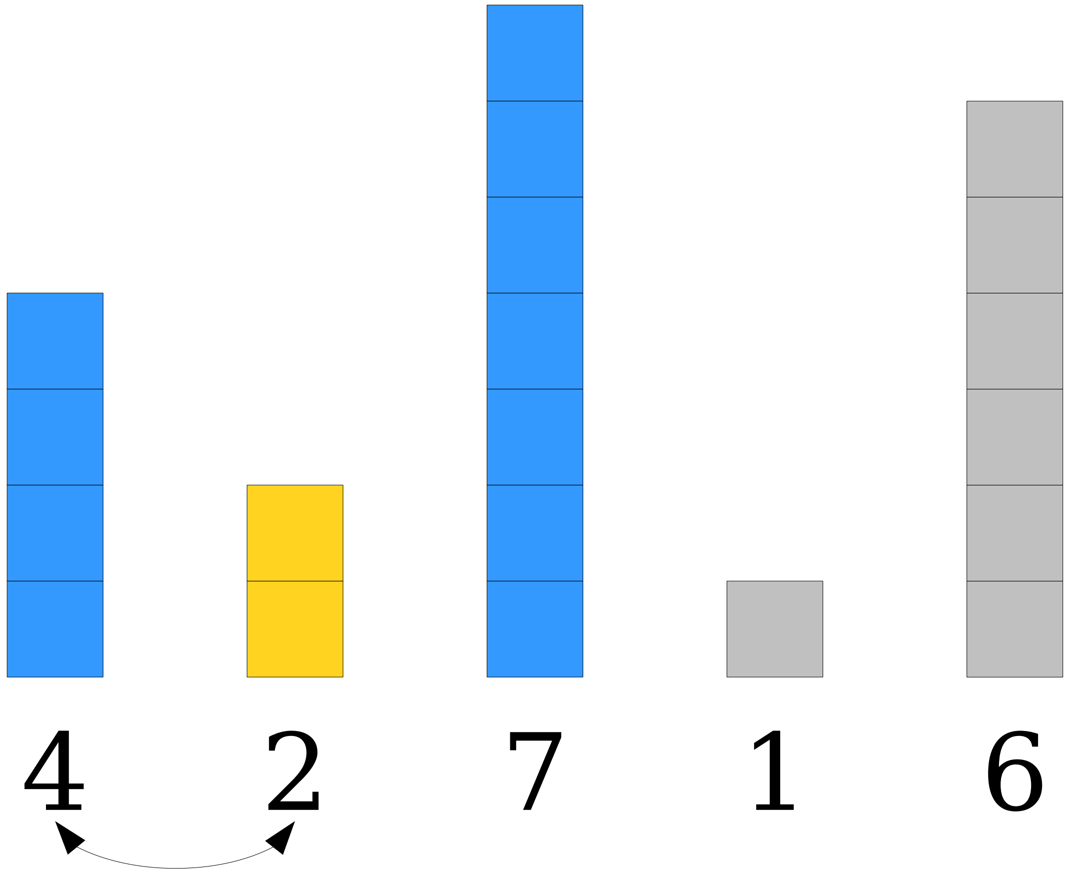
Our Next Idea: ***Insertion Sort***



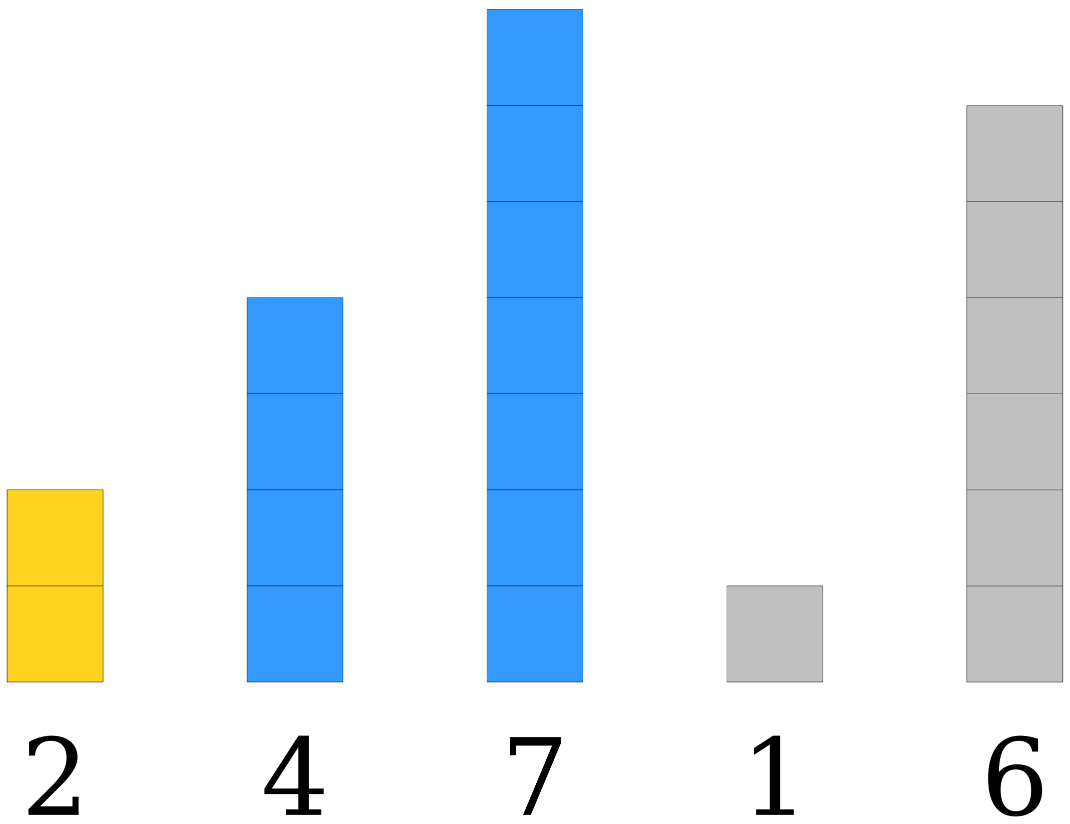
Our Next Idea: ***Insertion Sort***



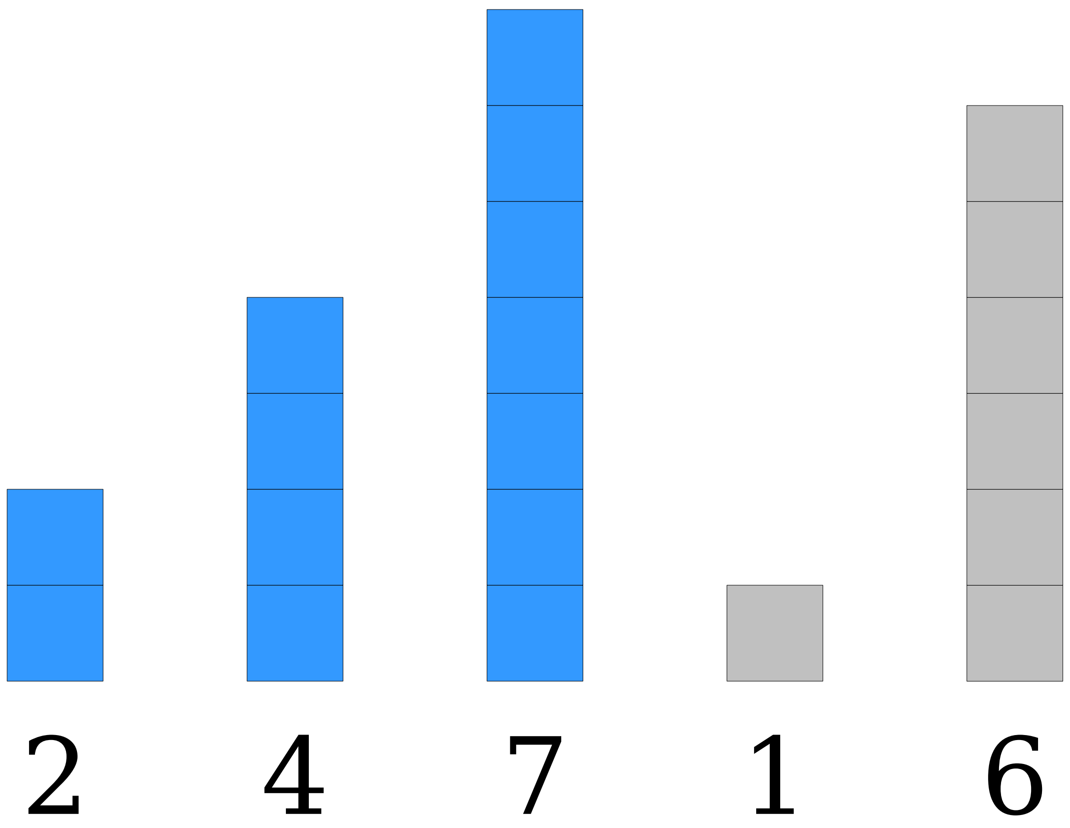
Our Next Idea: ***Insertion Sort***



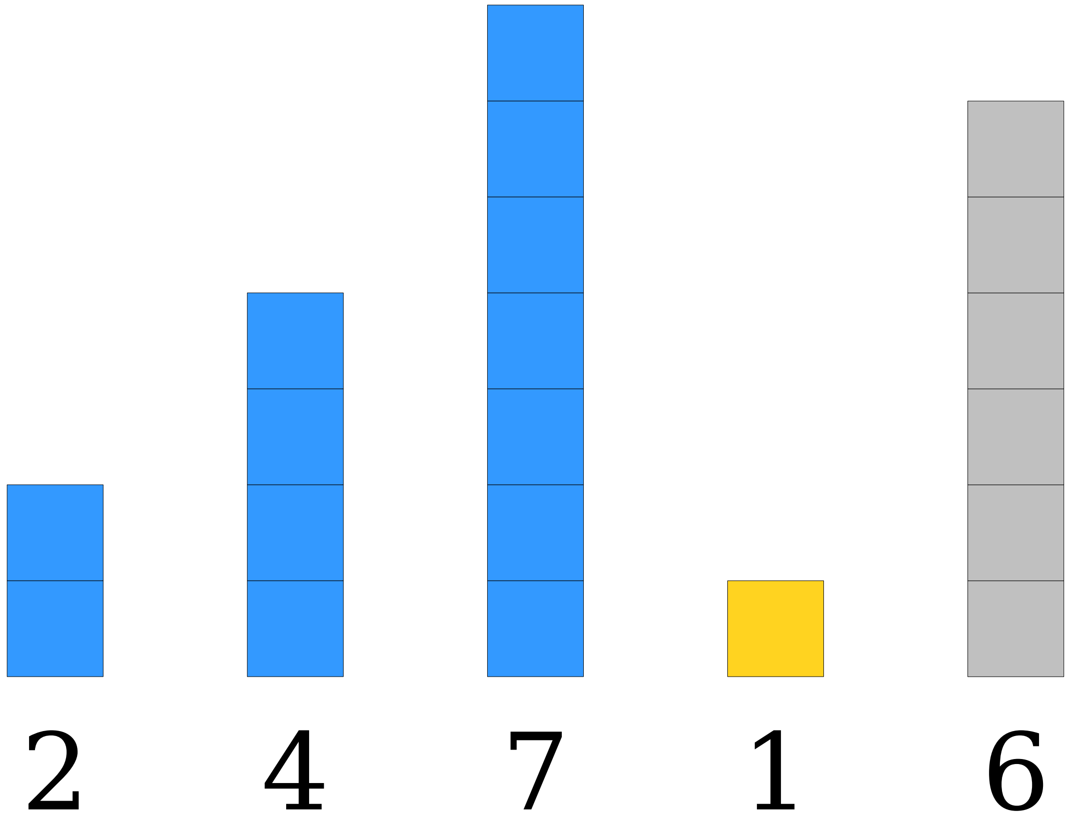
Our Next Idea: ***Insertion Sort***



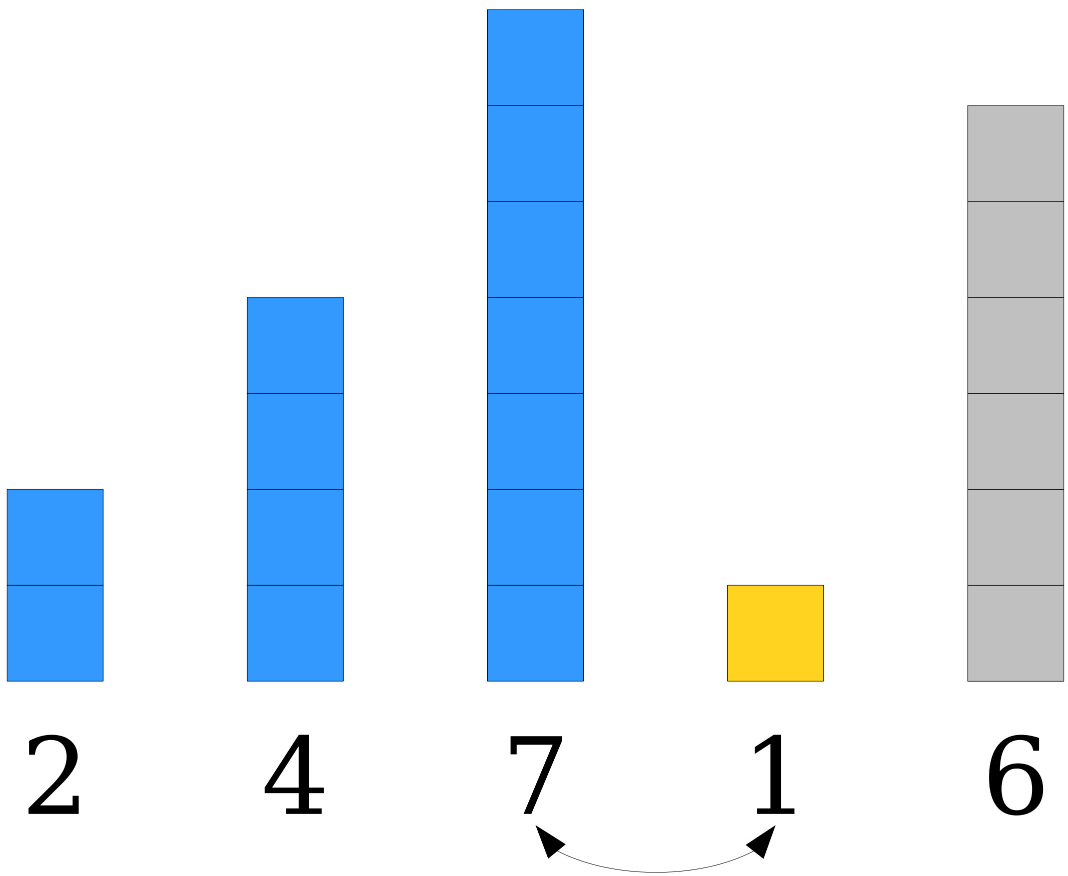
Our Next Idea: ***Insertion Sort***



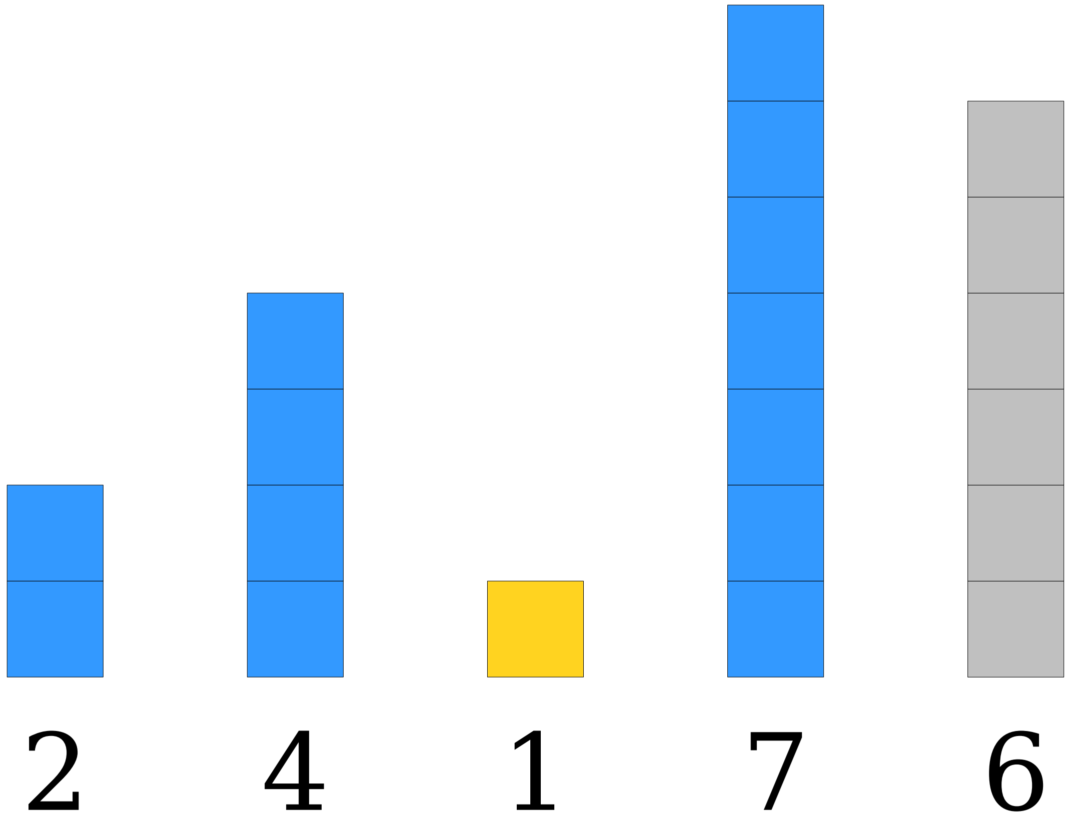
Our Next Idea: ***Insertion Sort***



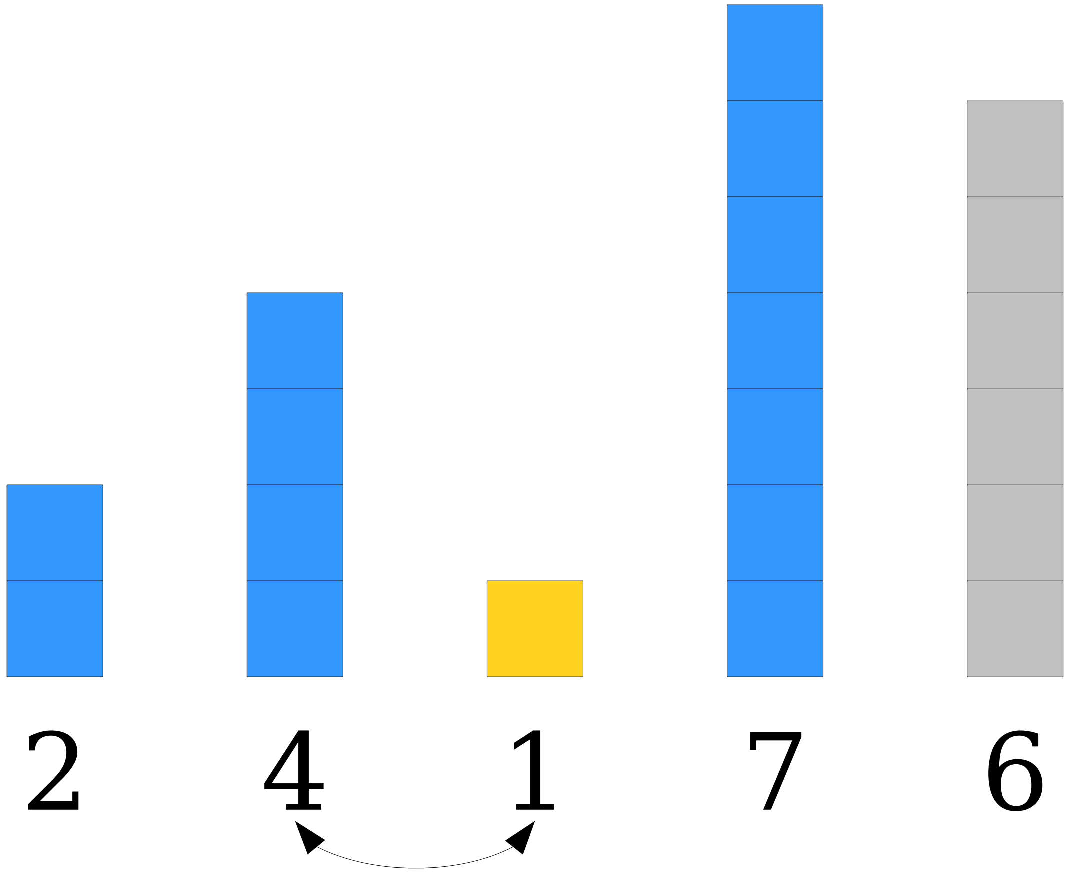
Our Next Idea: ***Insertion Sort***



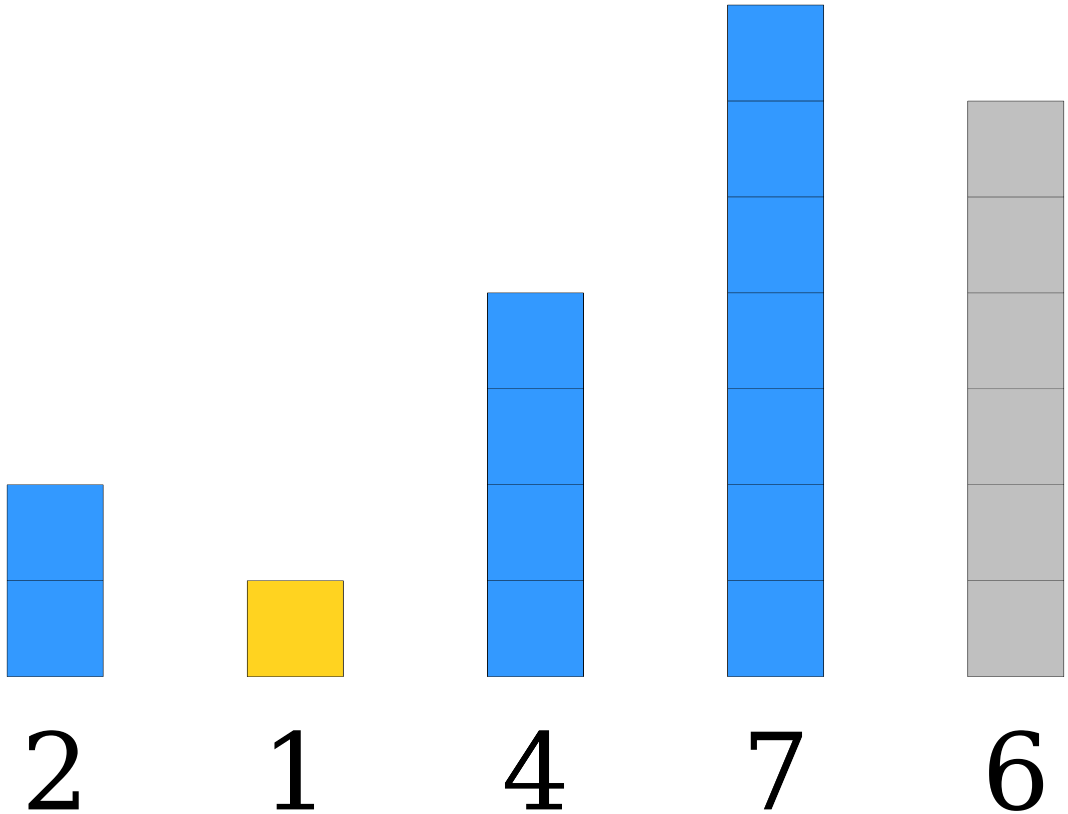
Our Next Idea: ***Insertion Sort***



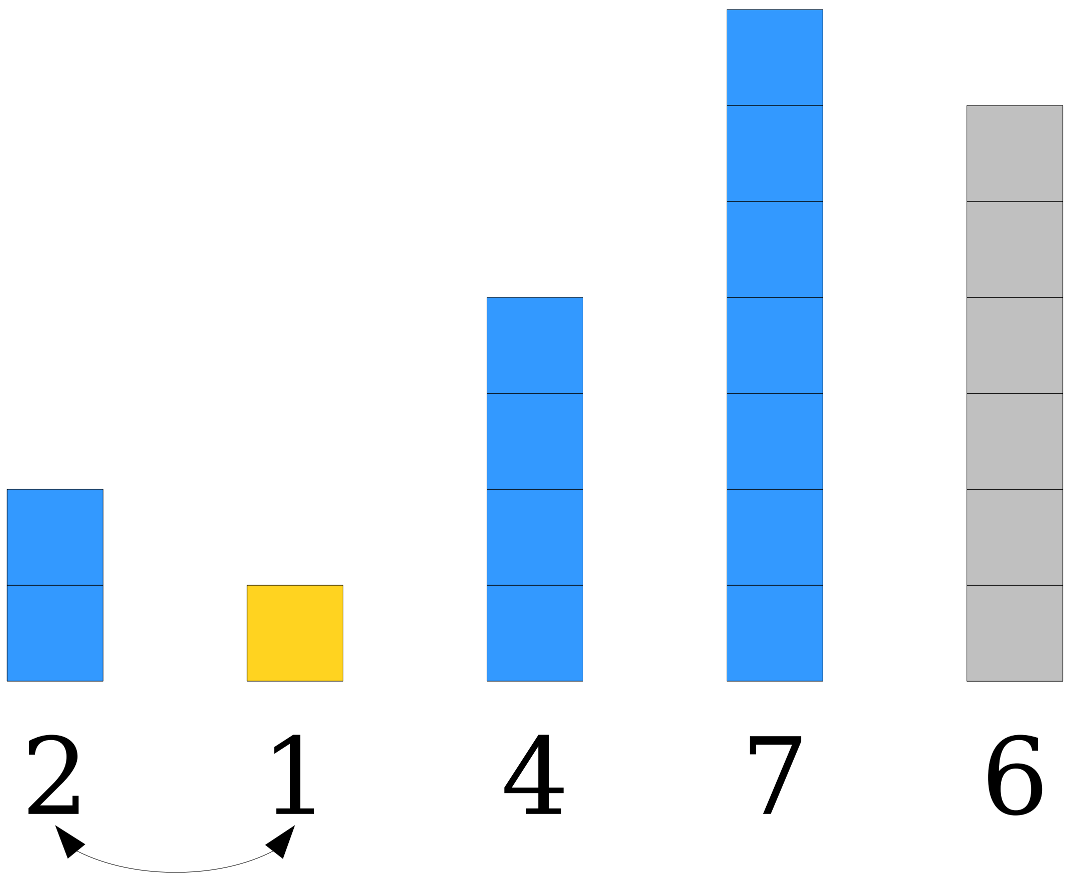
Our Next Idea: ***Insertion Sort***



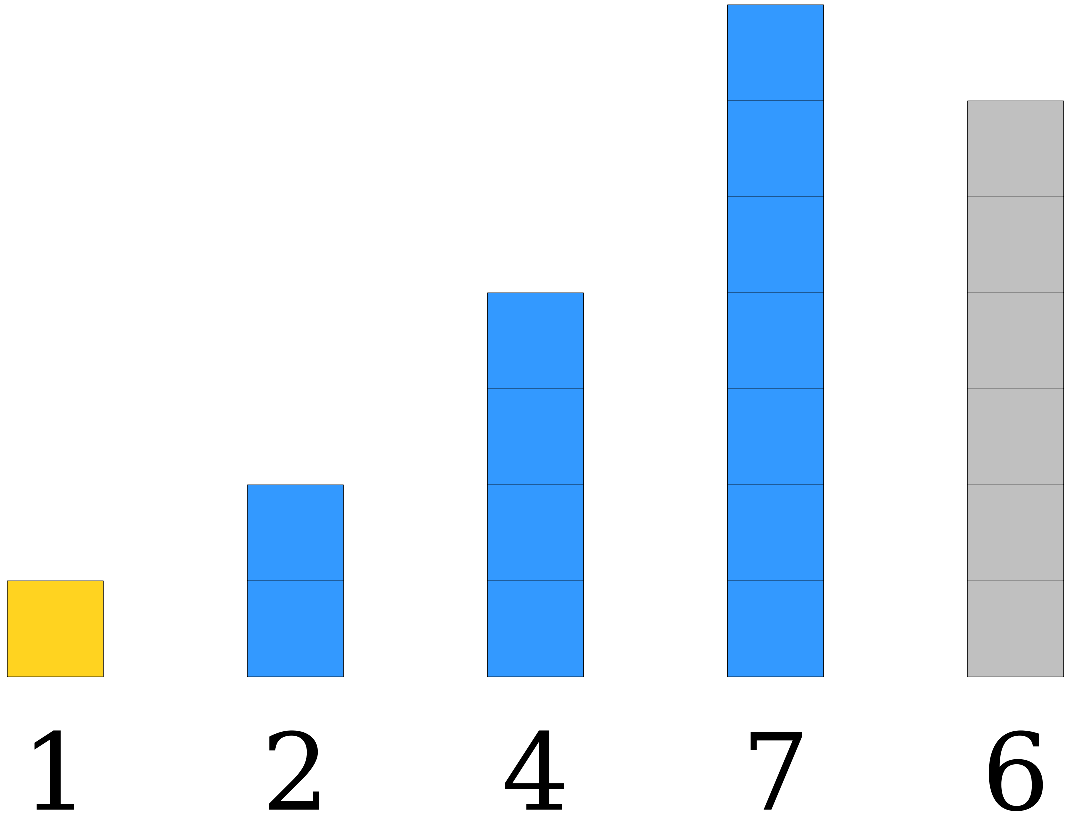
Our Next Idea: ***Insertion Sort***



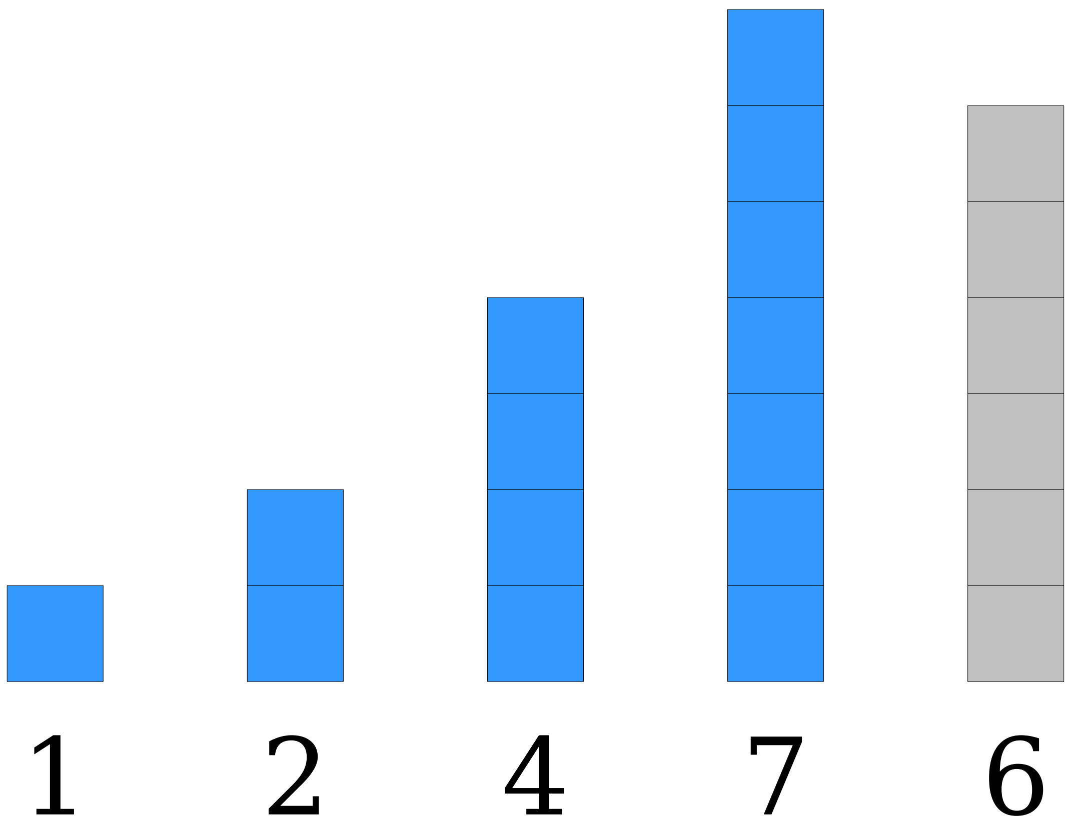
Our Next Idea: ***Insertion Sort***



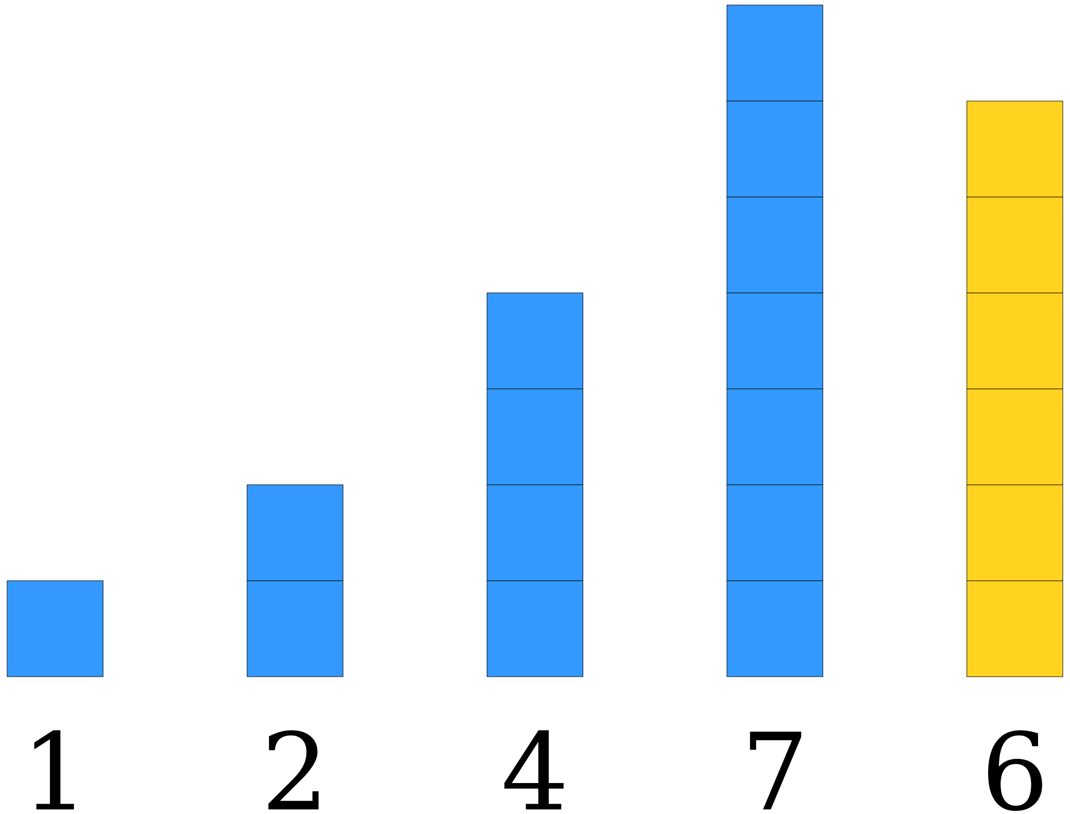
Our Next Idea: ***Insertion Sort***



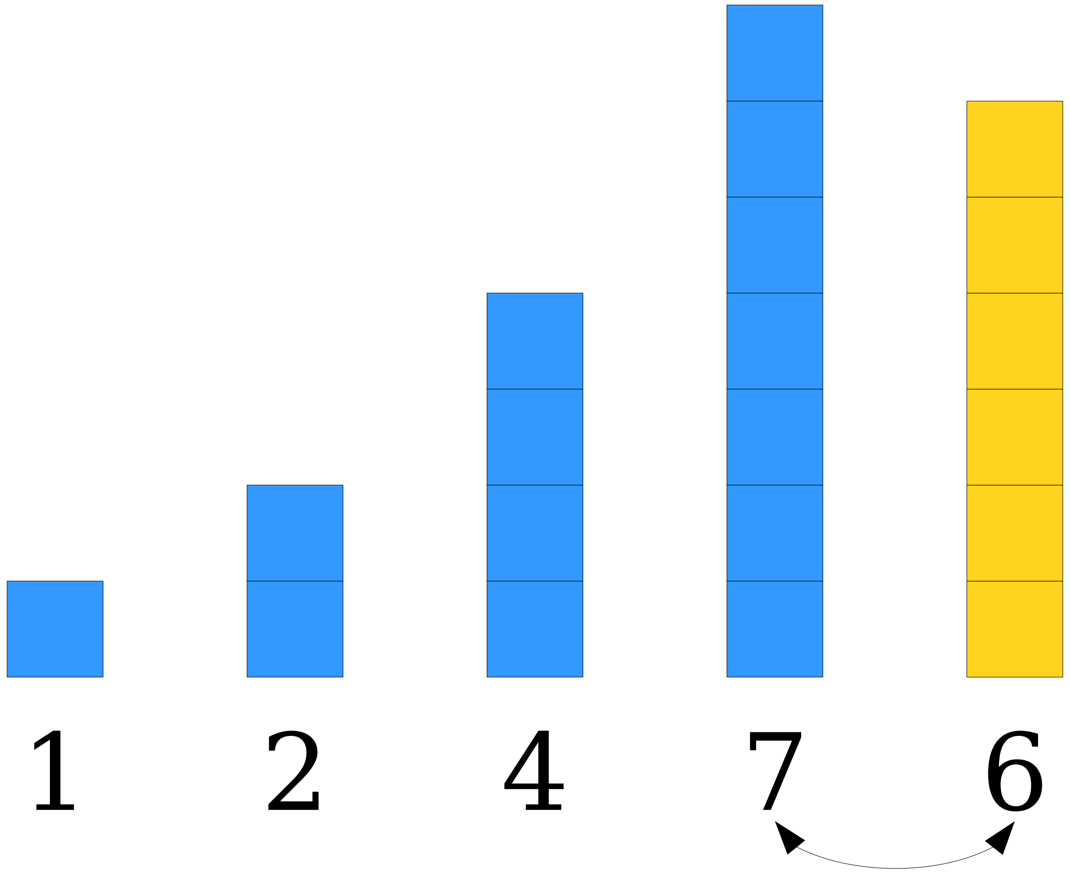
Our Next Idea: ***Insertion Sort***



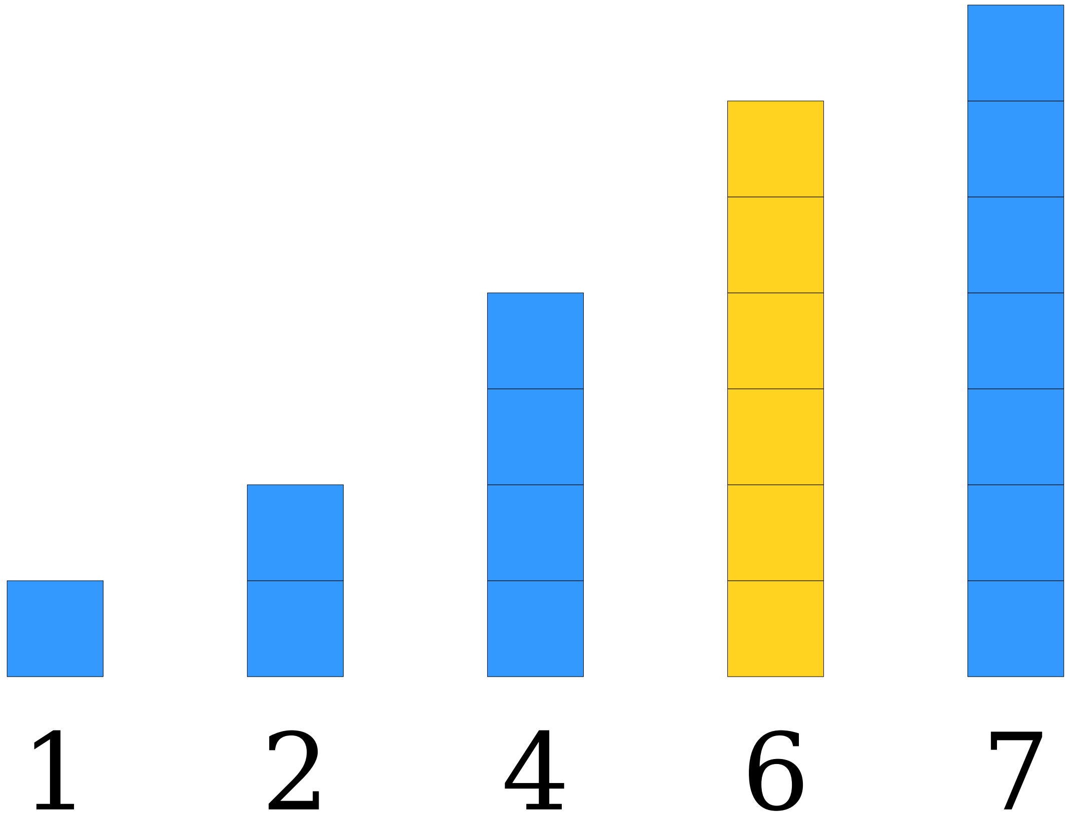
Our Next Idea: ***Insertion Sort***



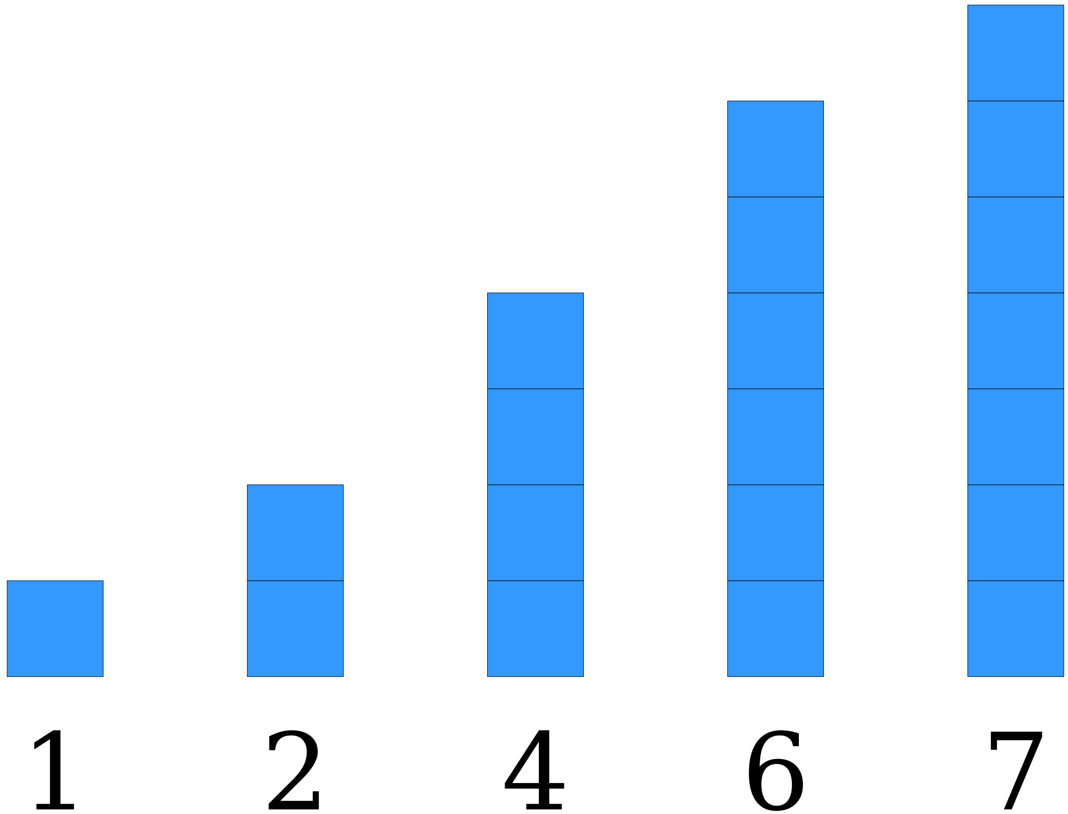
Our Next Idea: ***Insertion Sort***



Our Next Idea: ***Insertion Sort***



Our Next Idea: ***Insertion Sort***



Selection sort and insertion sort each run in time $O(n^2)$ in the worst case.

Doubling the size of the input quadruples the runtime.

Halving the size of the input quarters the runtime.

Thinking About O(n^2)

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

T(n)

14	6	3	9	7	16	2	15

T(½n)

5	10	8	11	1	13	12	4

T(½n)

Thinking About O(n^2)

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

$T(n)$

14	6	3	9	7	16	2	15

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

5	10	8	11	1	13	12	4

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

Thinking About O(n^2)

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

$\mathrm{T}(n)$

2	3	6	7	9	14	15	16

$\tfrac{1}{4}\mathrm{T}(n)$

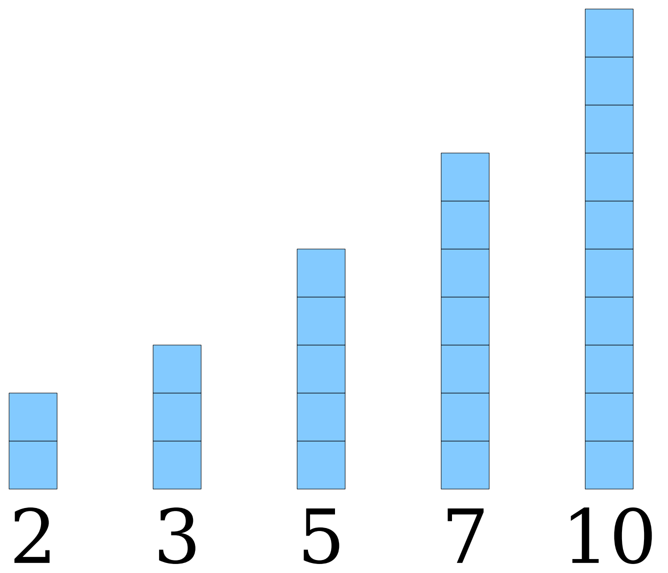
1	4	5	8	10	11	12	13

$\tfrac{1}{4}\mathrm{T}(n)$

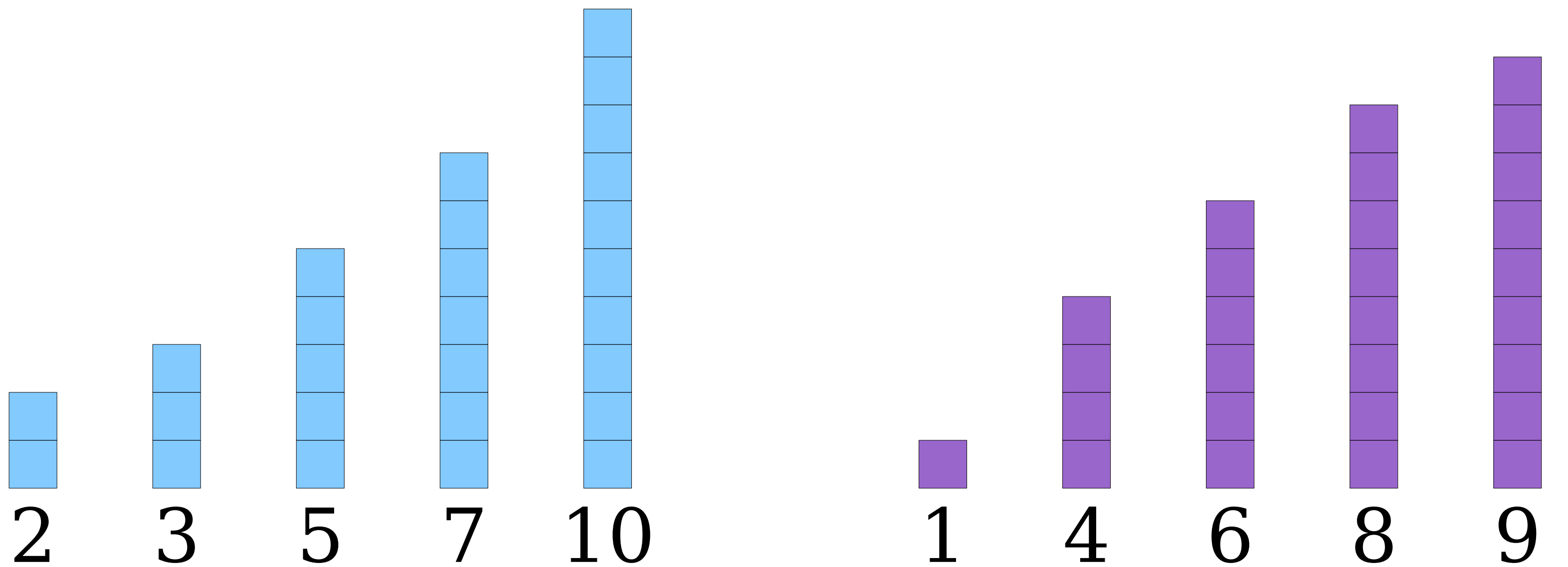
$$2 \cdot \tfrac{1}{4}\mathrm{T}(n) = \tfrac{1}{2}\mathrm{T}(n)$$

The Key Insight: ***Merge***

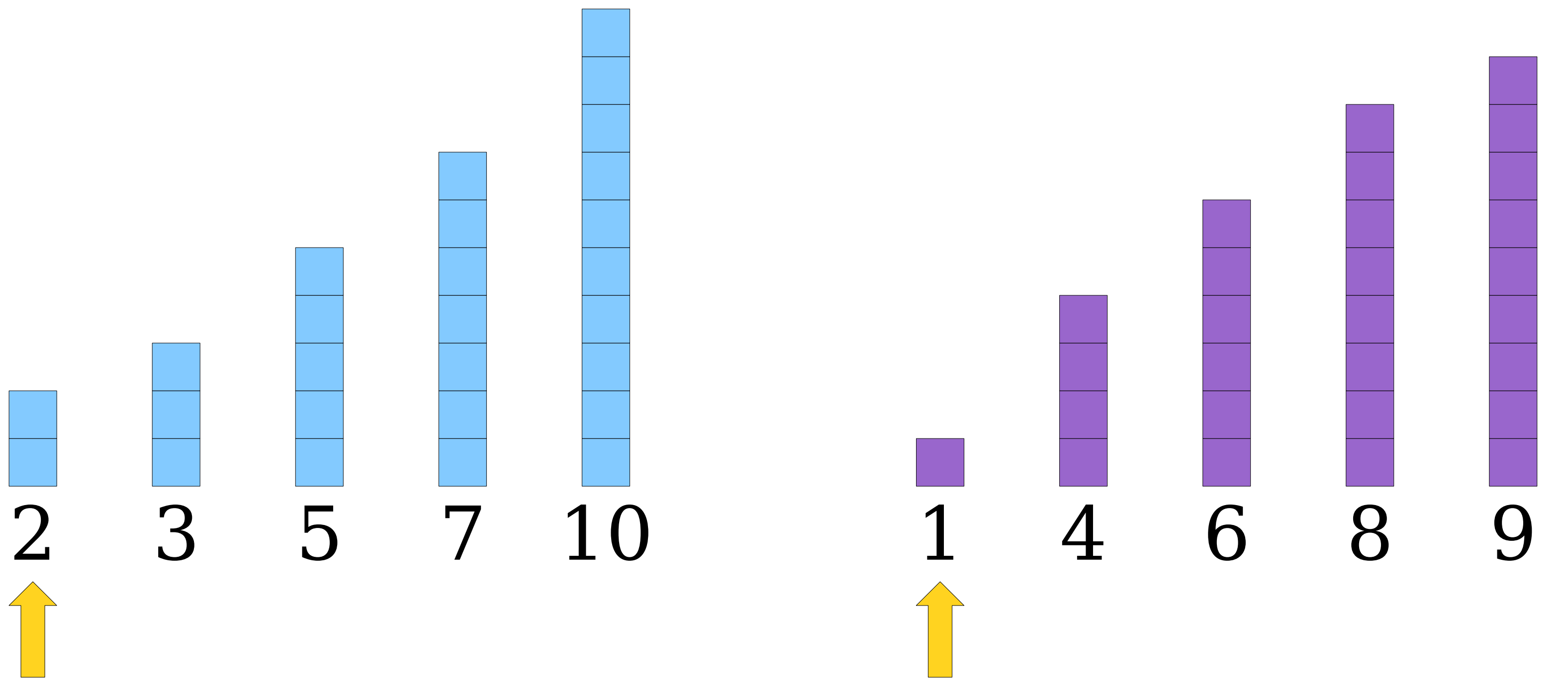
The Key Insight: ***Merge***



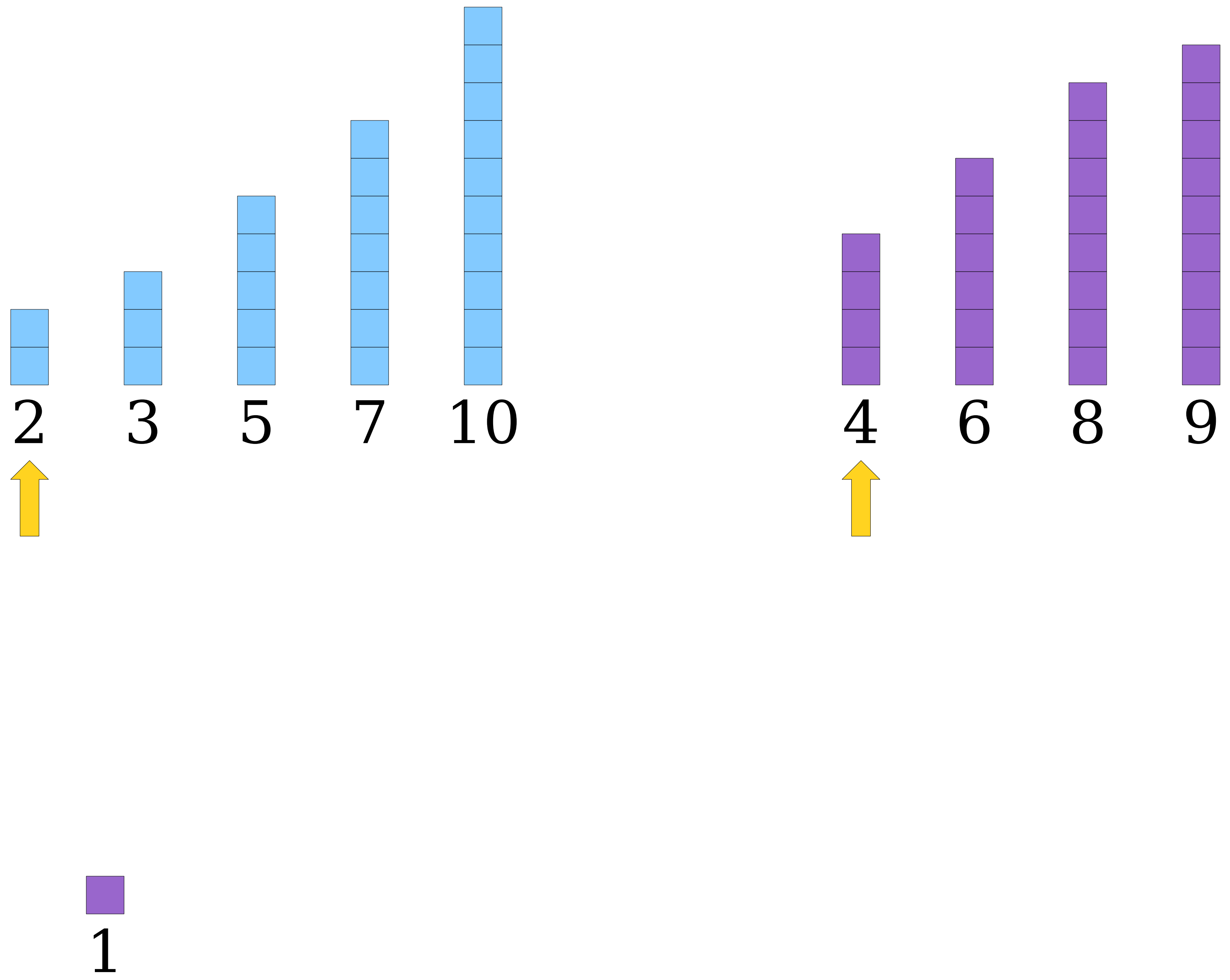
The Key Insight: ***Merge***



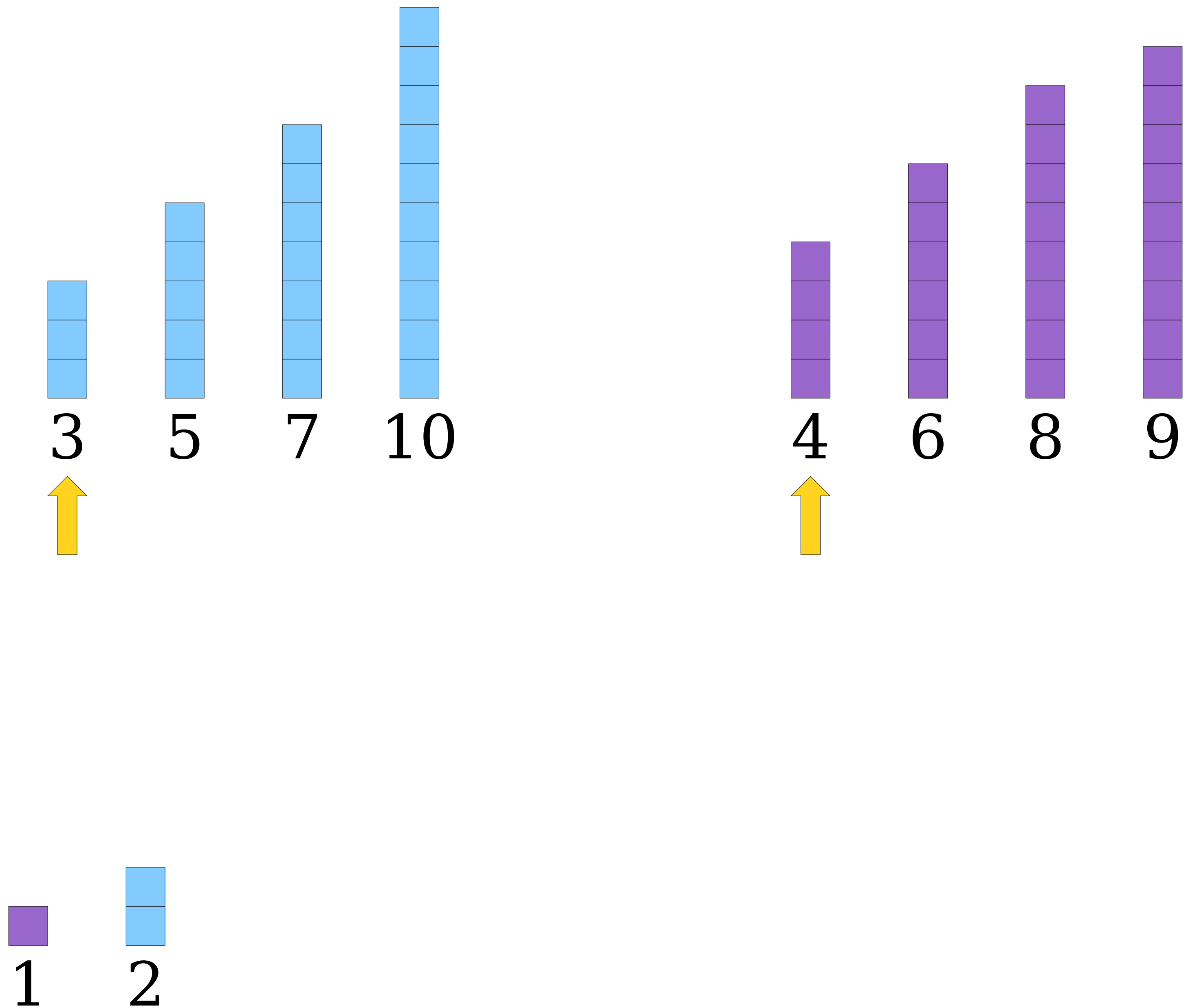
The Key Insight: ***Merge***



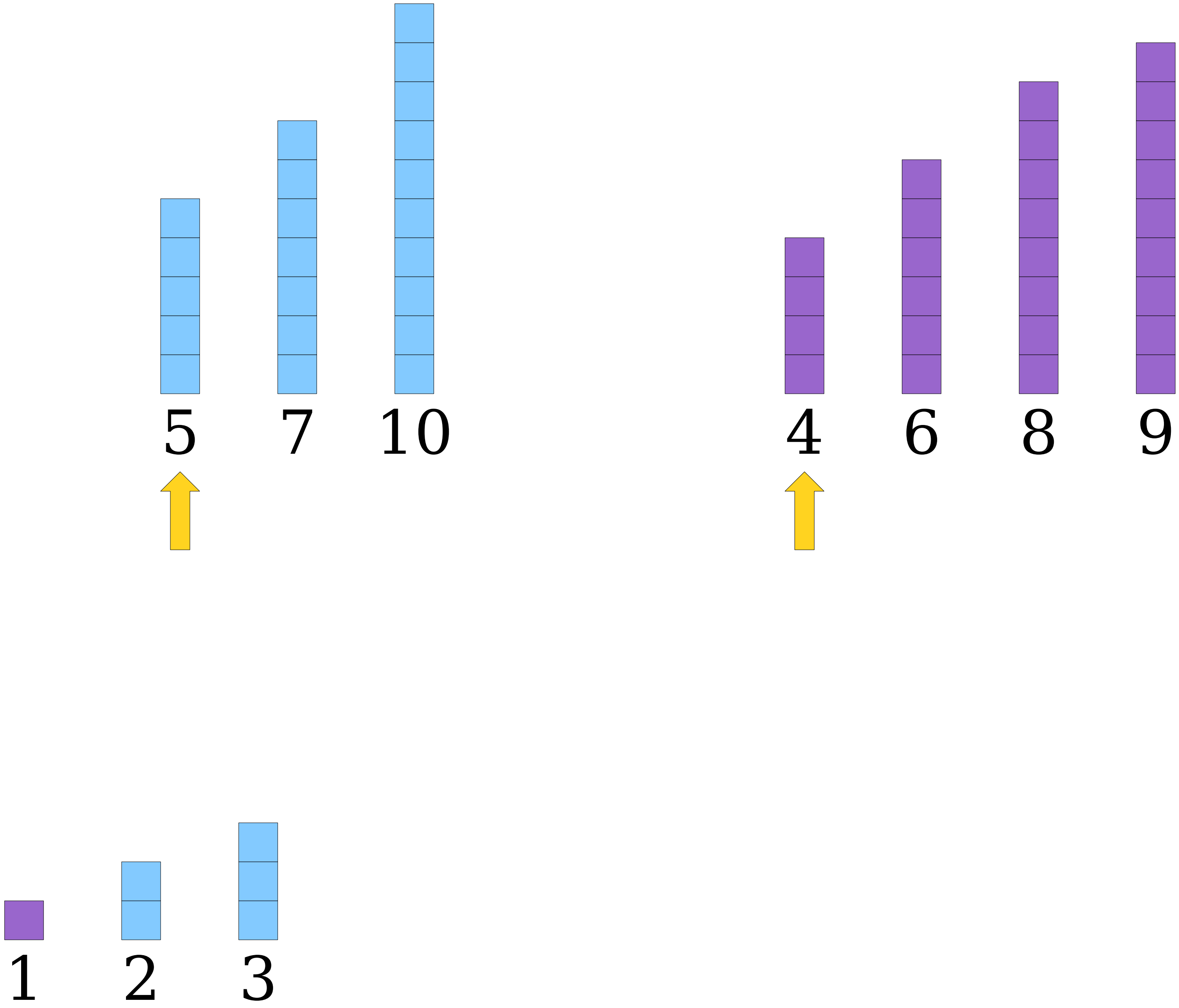
The Key Insight: ***Merge***



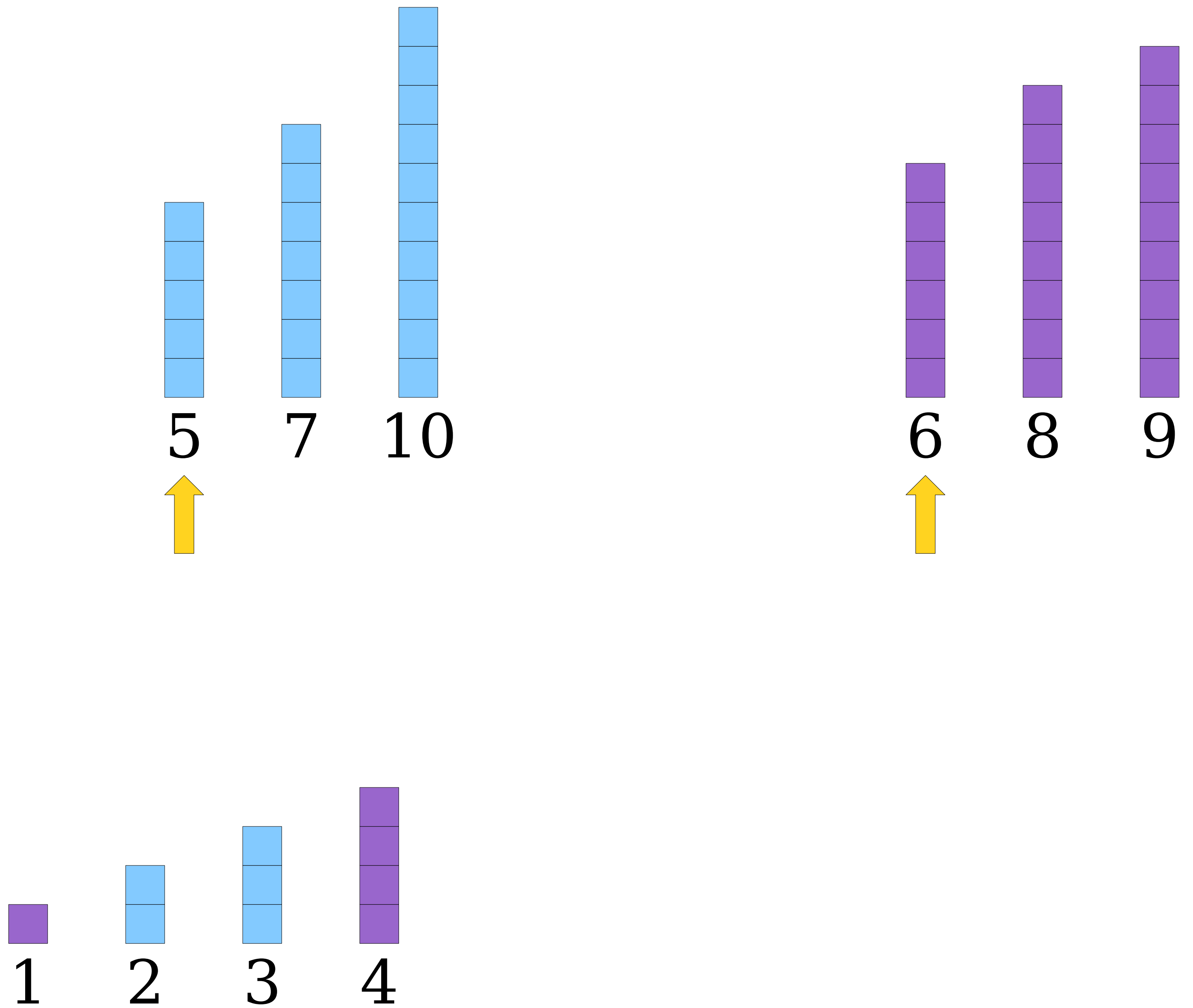
The Key Insight: ***Merge***



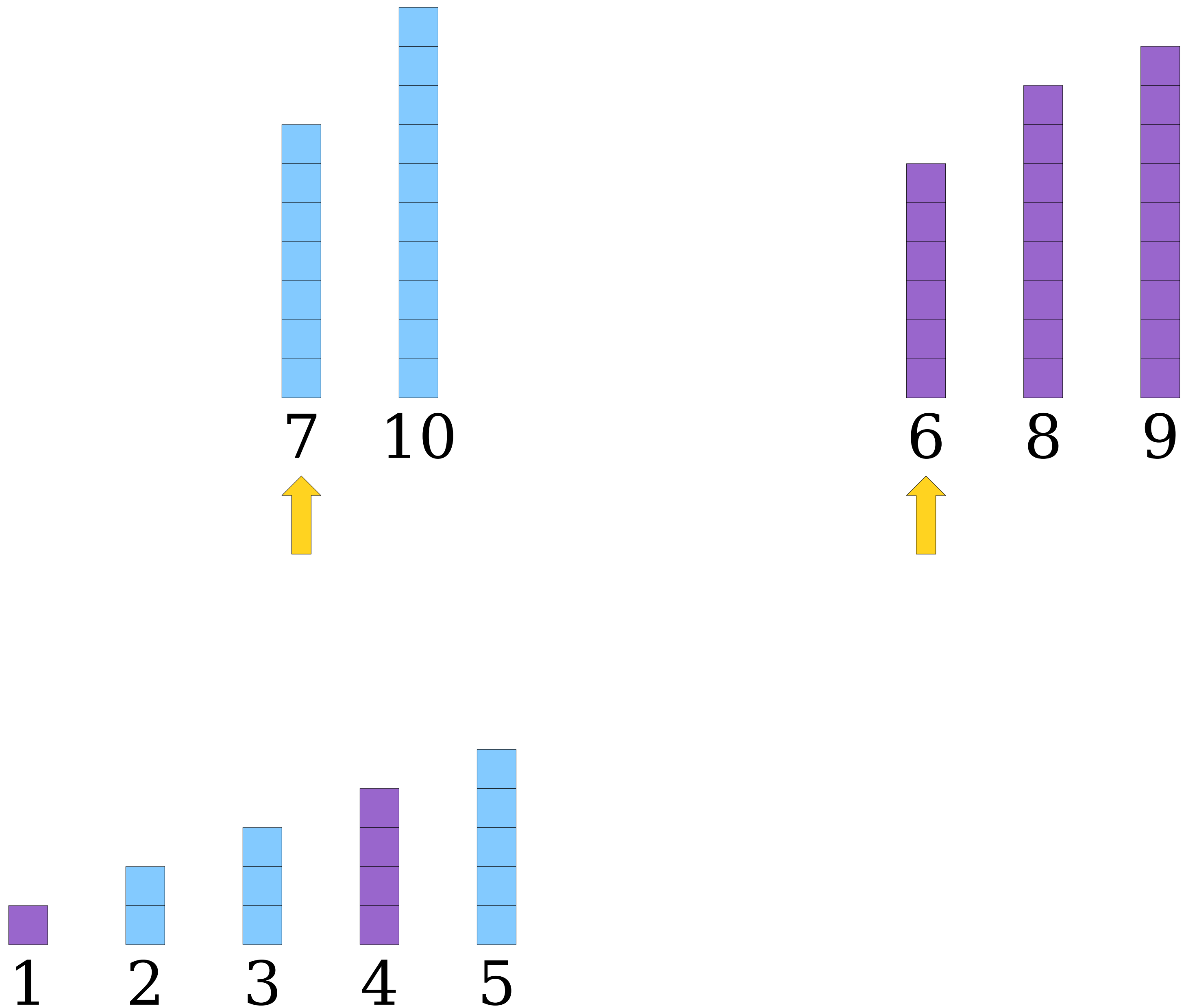
The Key Insight: ***Merge***



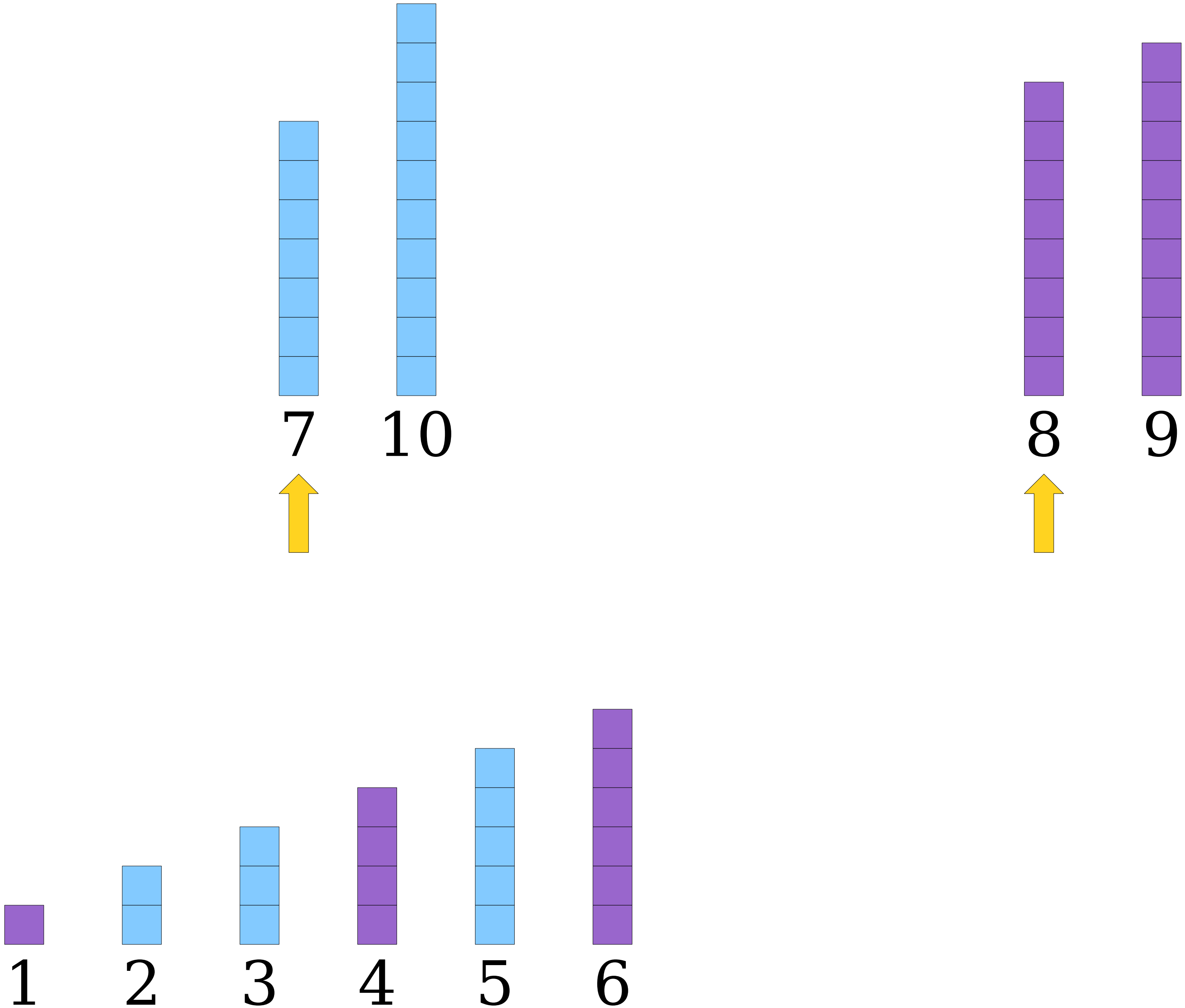
The Key Insight: ***Merge***



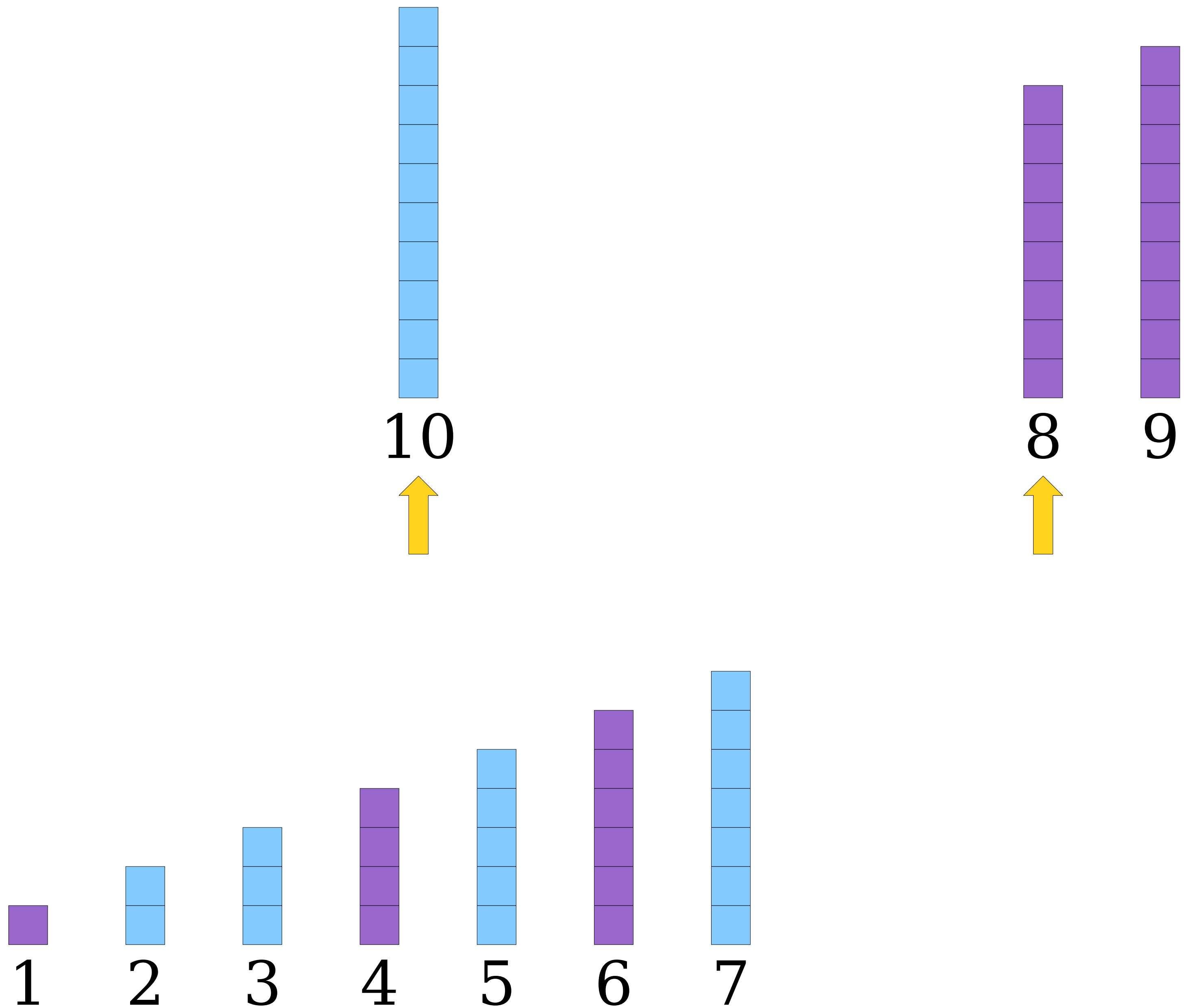
The Key Insight: ***Merge***



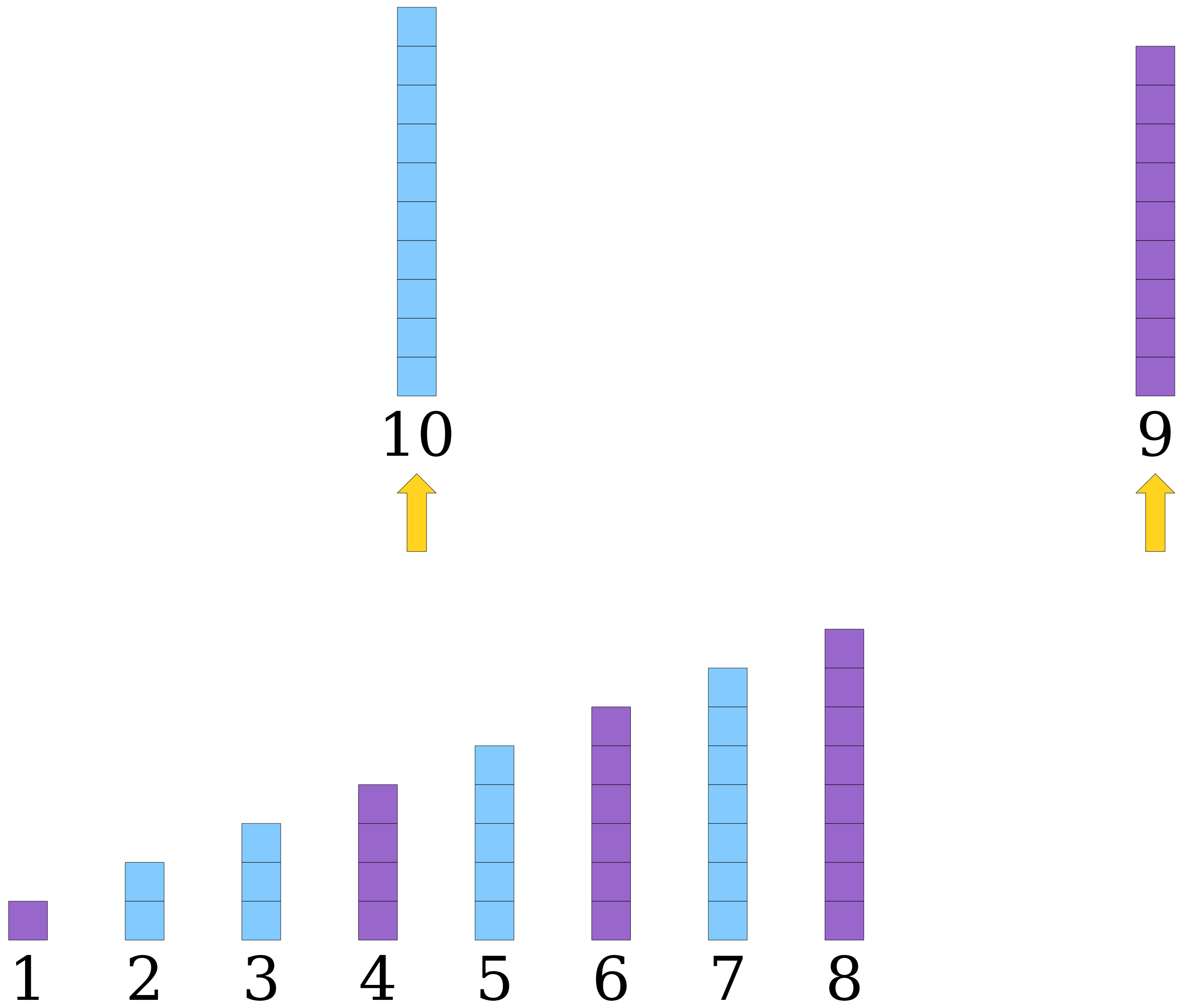
The Key Insight: ***Merge***



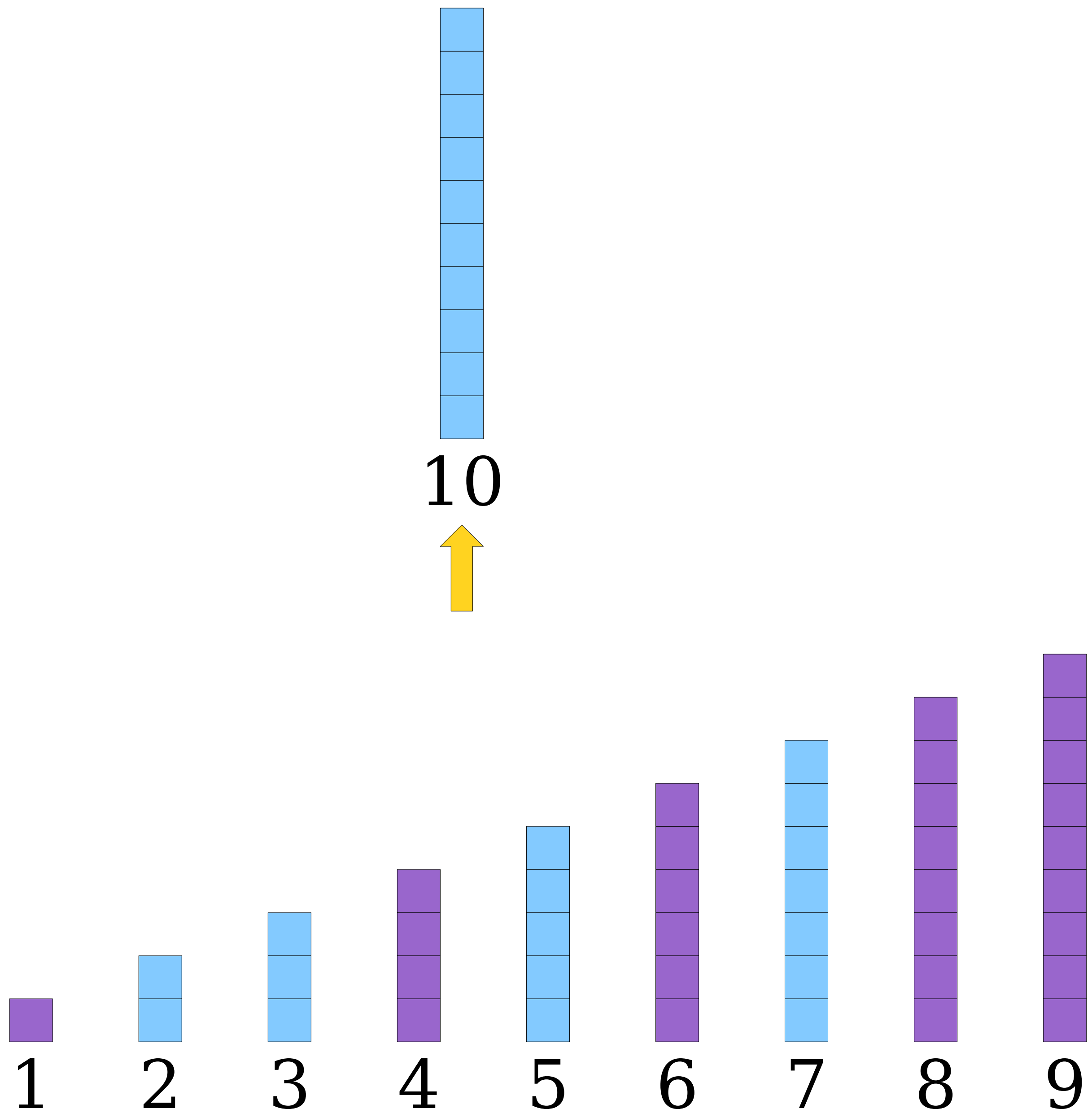
The Key Insight: ***Merge***



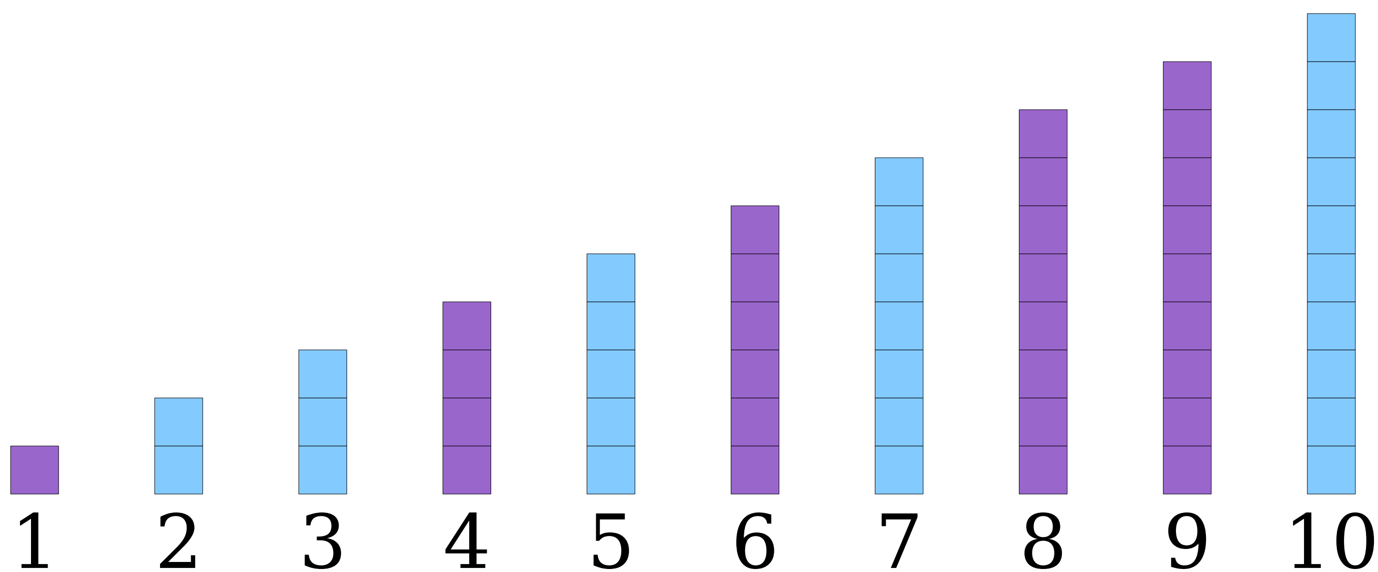
The Key Insight: ***Merge***



The Key Insight: ***Merge***



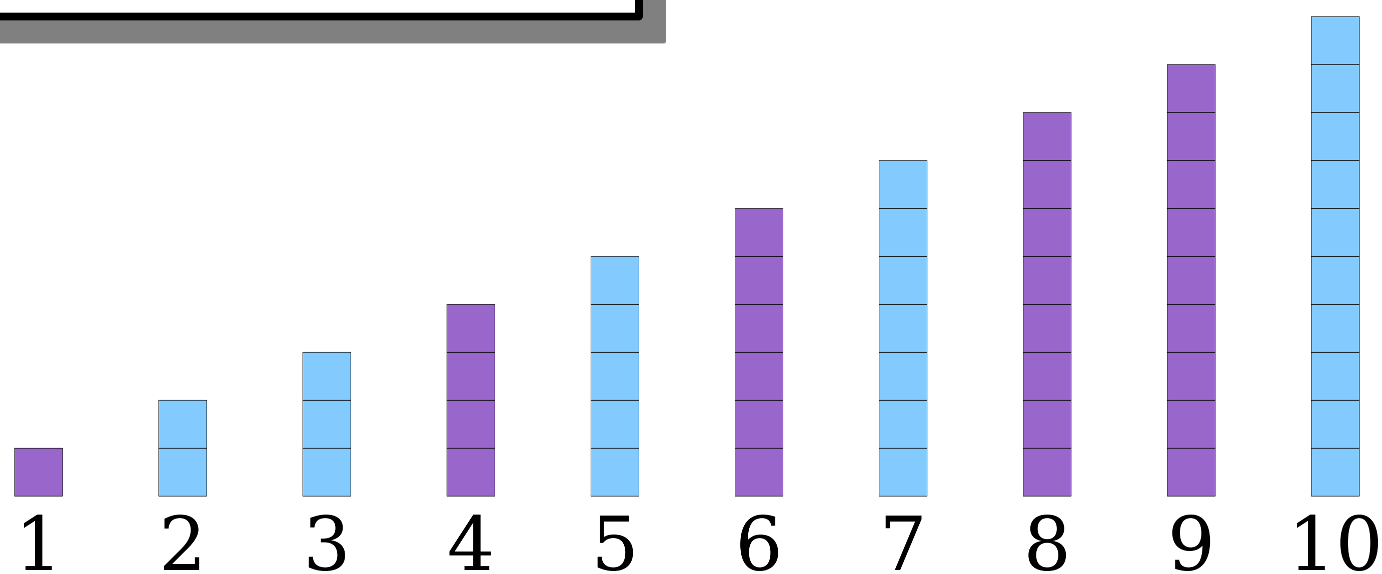
The Key Insight: ***Merge***



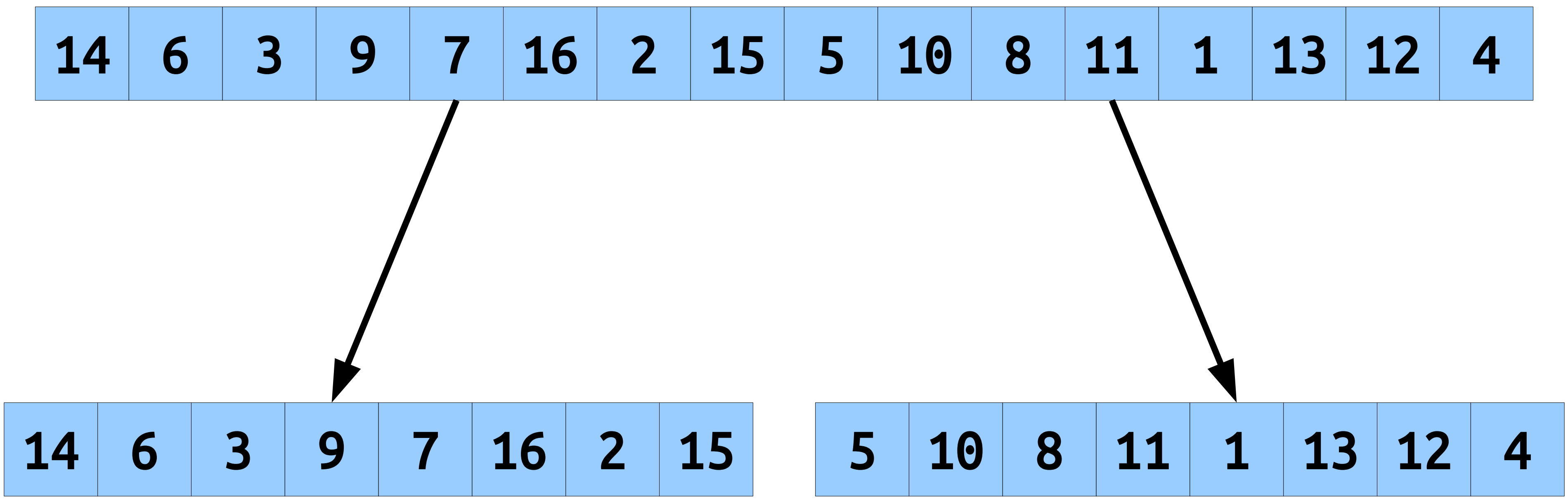
The Key Insight: ***Merge***

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”

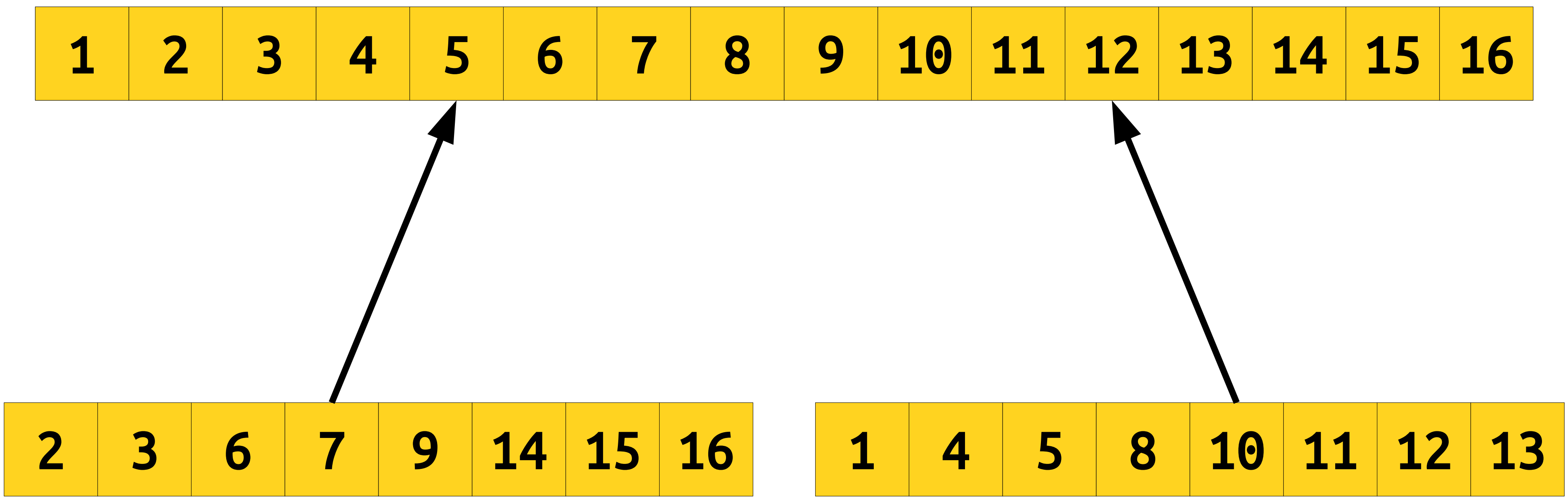
14	6	3	9	7	16	2	15	5	10	8	11	1	13	12	4

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.

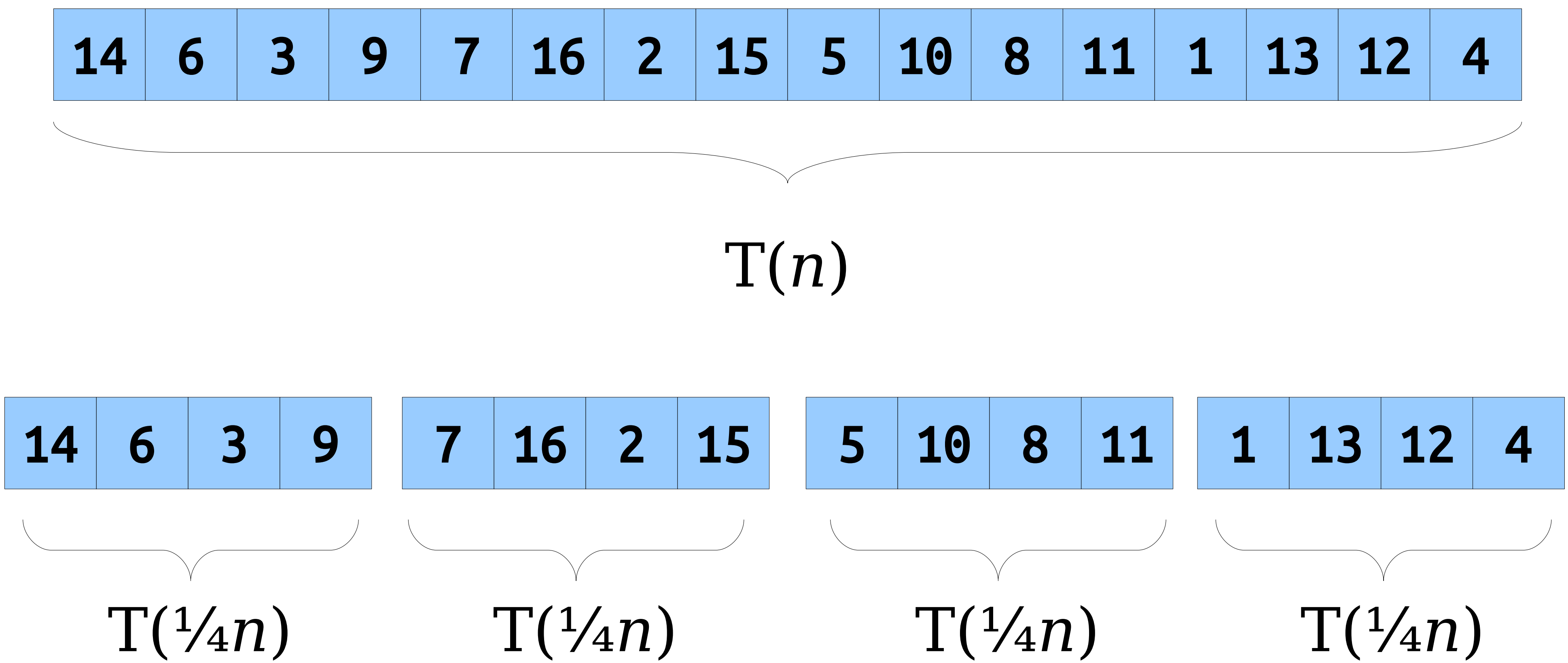
“Split Sort”



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

New Stuff!

“Double Split Sort”



"Double Split Sort"

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

$T(n)$

14	6	3	9

${}^{1}/_{16}\ T(n)$

7	16	2	15

${}^{1}/_{16}\ T(n)$

5	10	8	11

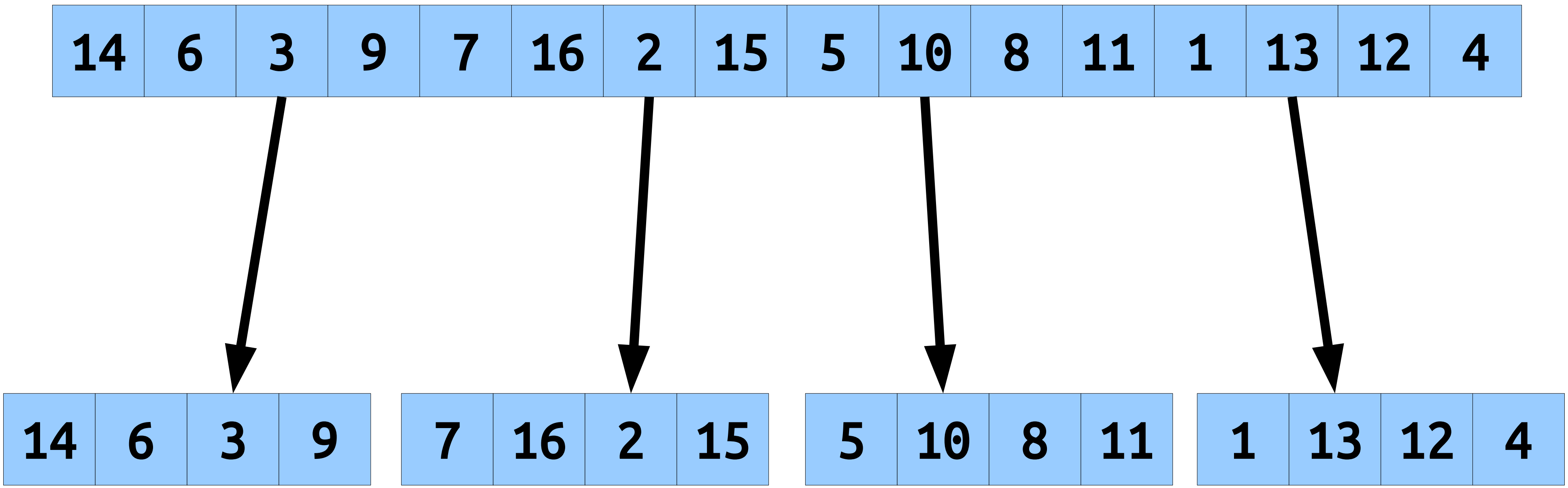
${}^{1}/_{16}\ T(n)$

1	13	12	4

${}^{1}/_{16}\ T(n)$

$$4 \cdot {}^{1}/_{16}\ T(n) \quad = \quad \boldsymbol{{}^{1}\!/_{4}T(n)}$$

“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

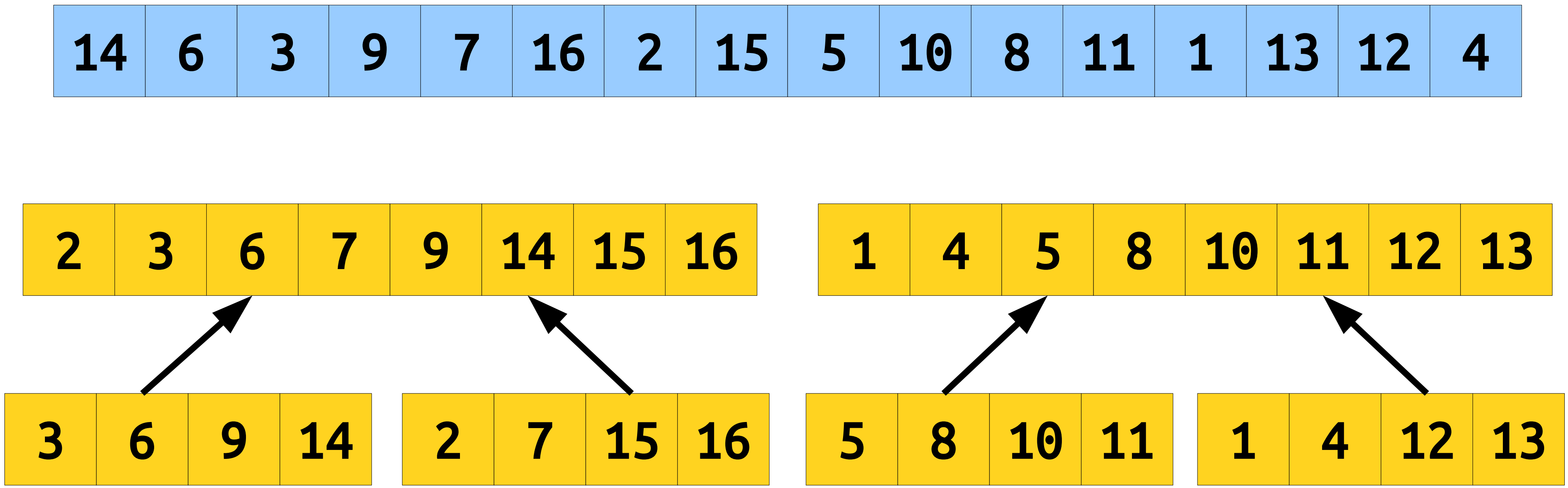
2	7	15	16

5	8	10	11

1	4	12	13

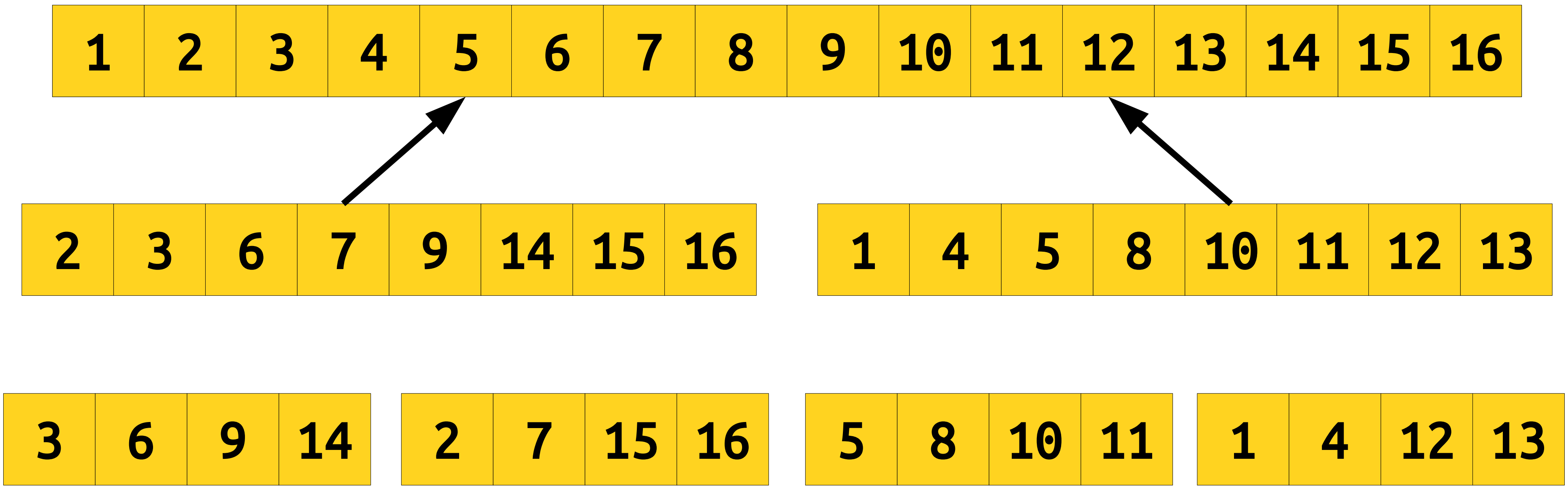
1. Split the input into quarters.
2. Insertion sort each quarter.

“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”

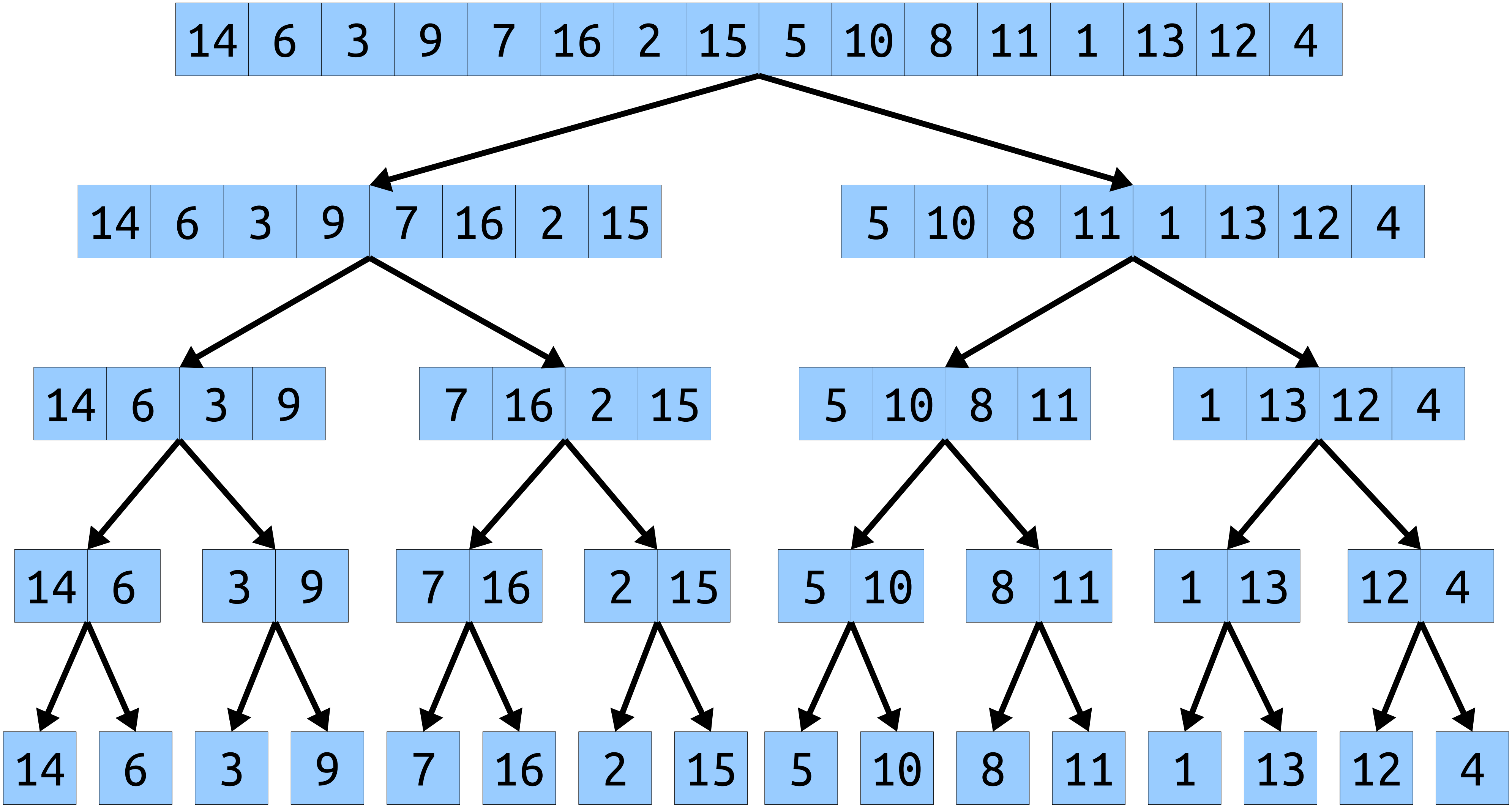


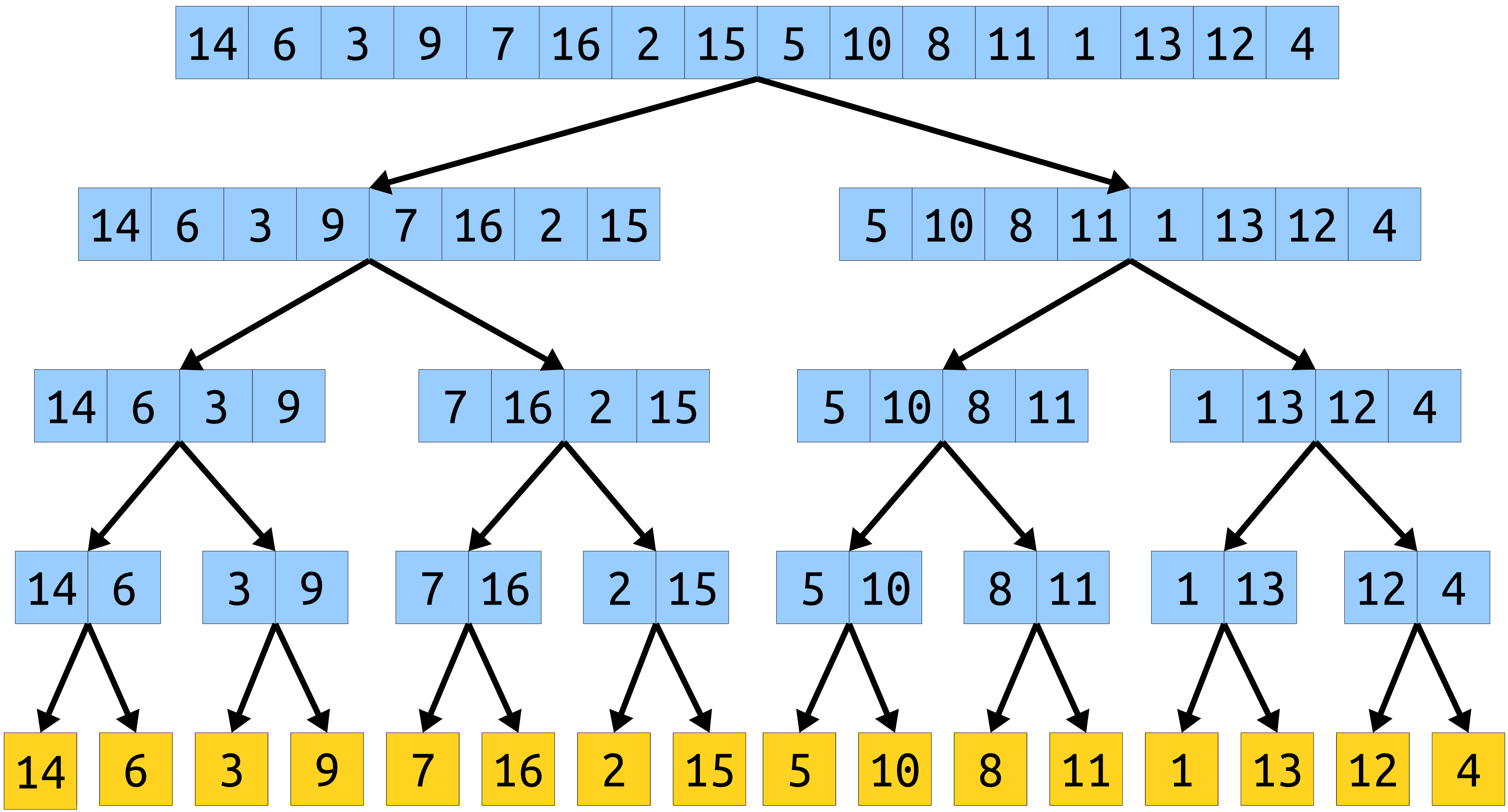
1. Split the input into quarters.
2. Insertion sort each quarter.
3. Merge two pairs of quarters into halves.
4. Merge the two halves back together.

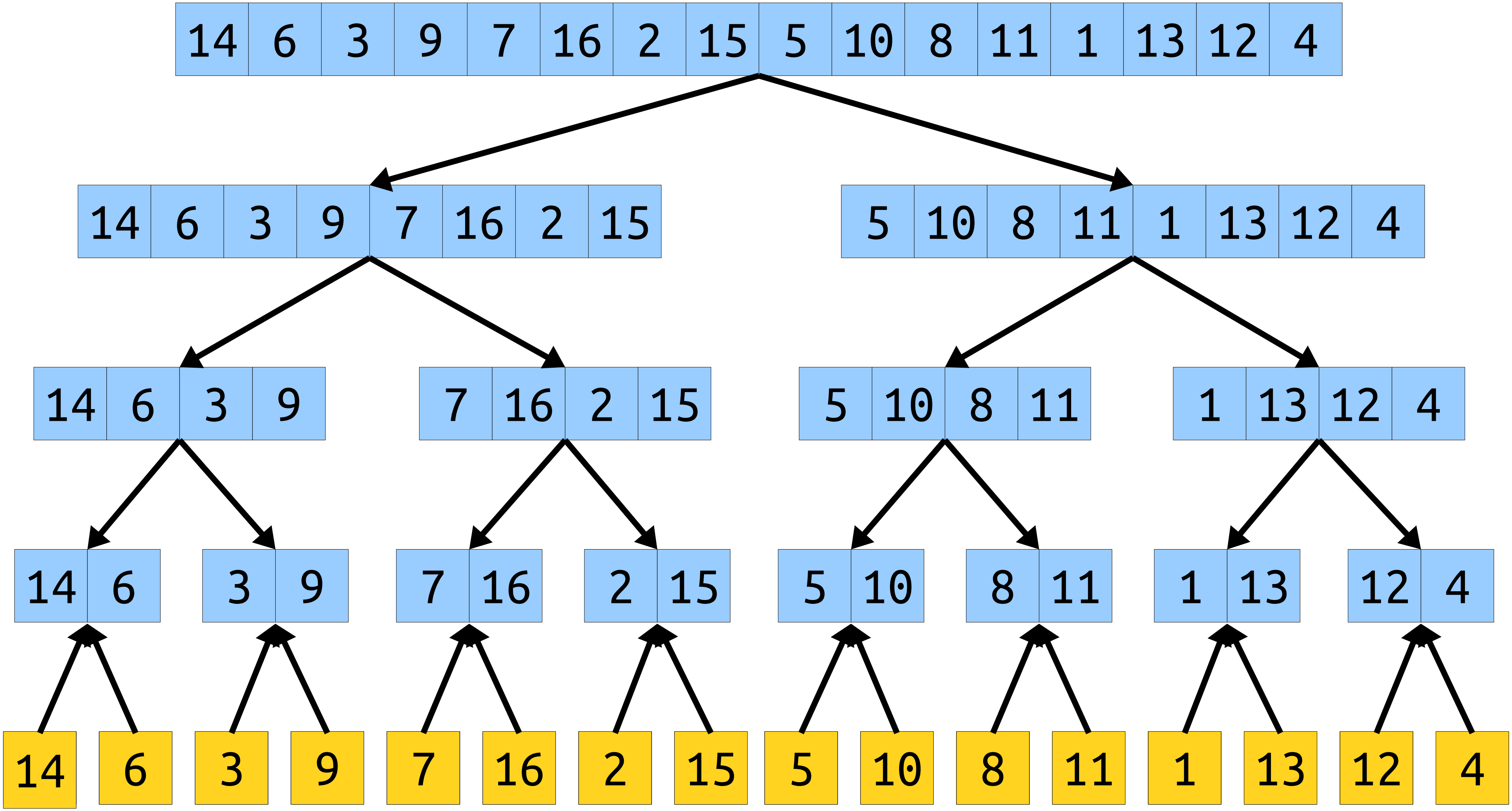
Prediction: This should be four times as fast as insertion sort.

Splitting to the Extreme

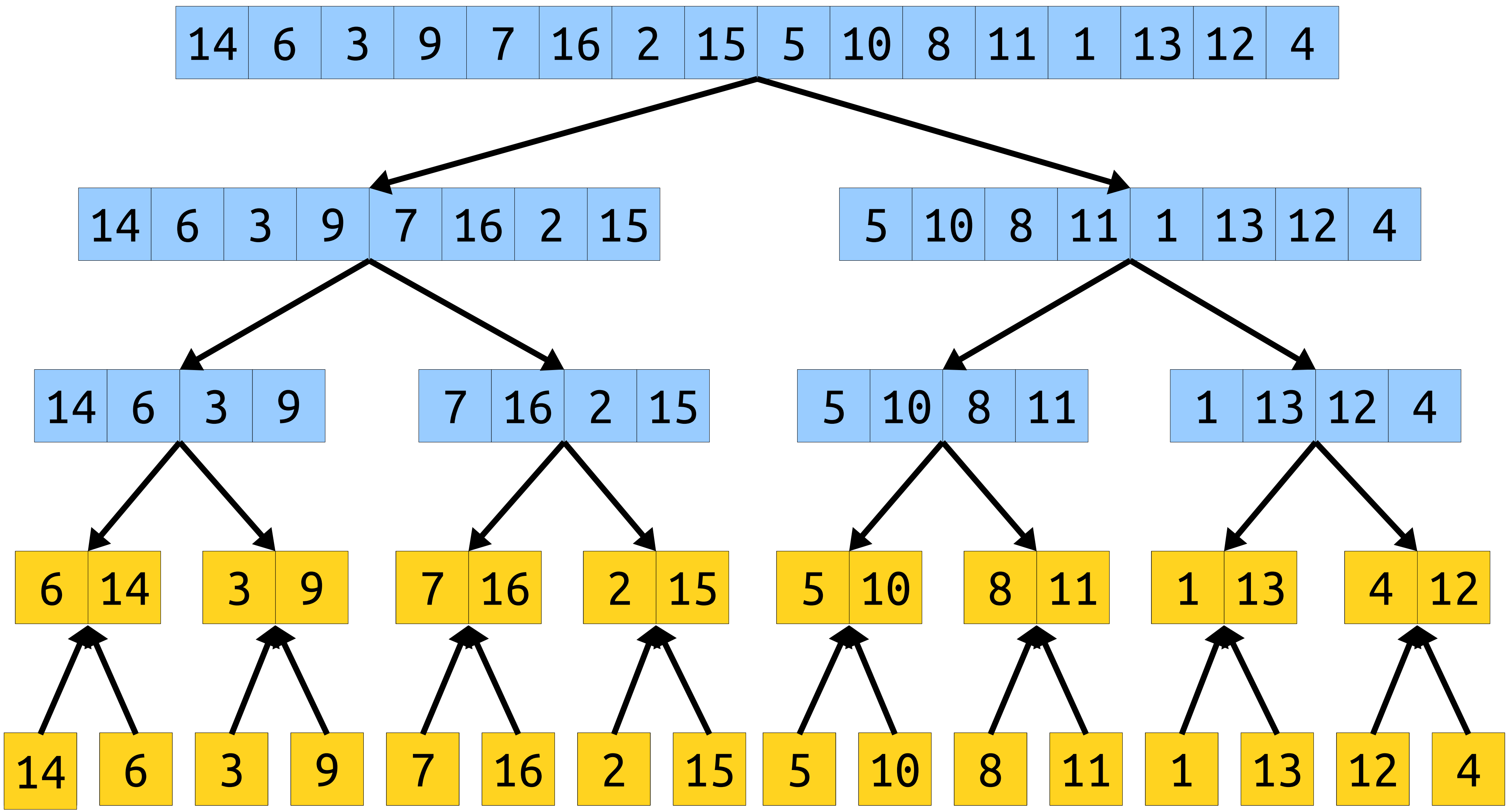
- Splitting our array in half, sorting each half, and merging the halves was twice as fast as insertion sort.
- Splitting our array in quarters, sorting each quarter, and merging the quarters was four times as fast as insertion sort.
- ***Question:*** What happens if we *never stop splitting?*


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

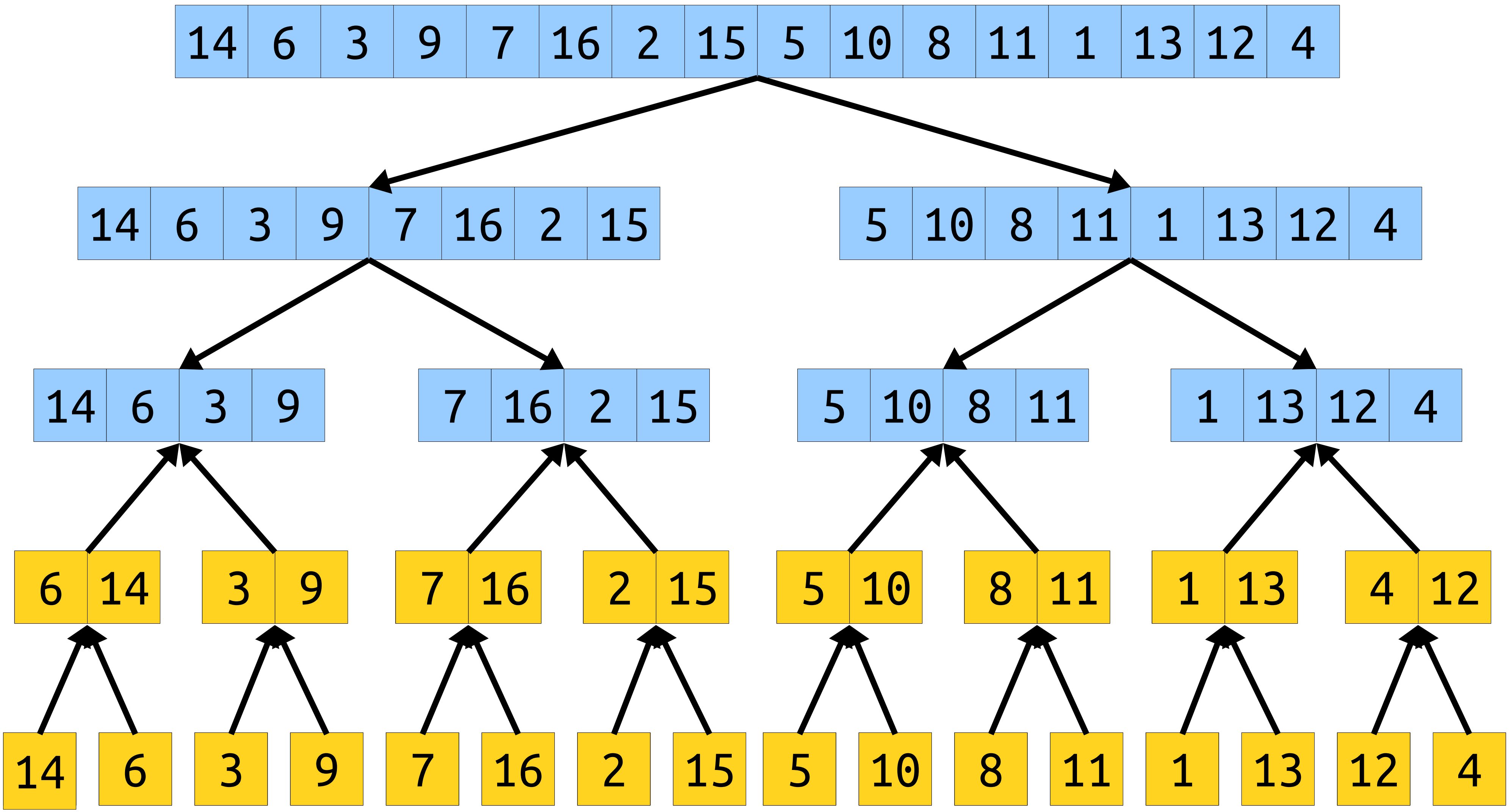

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

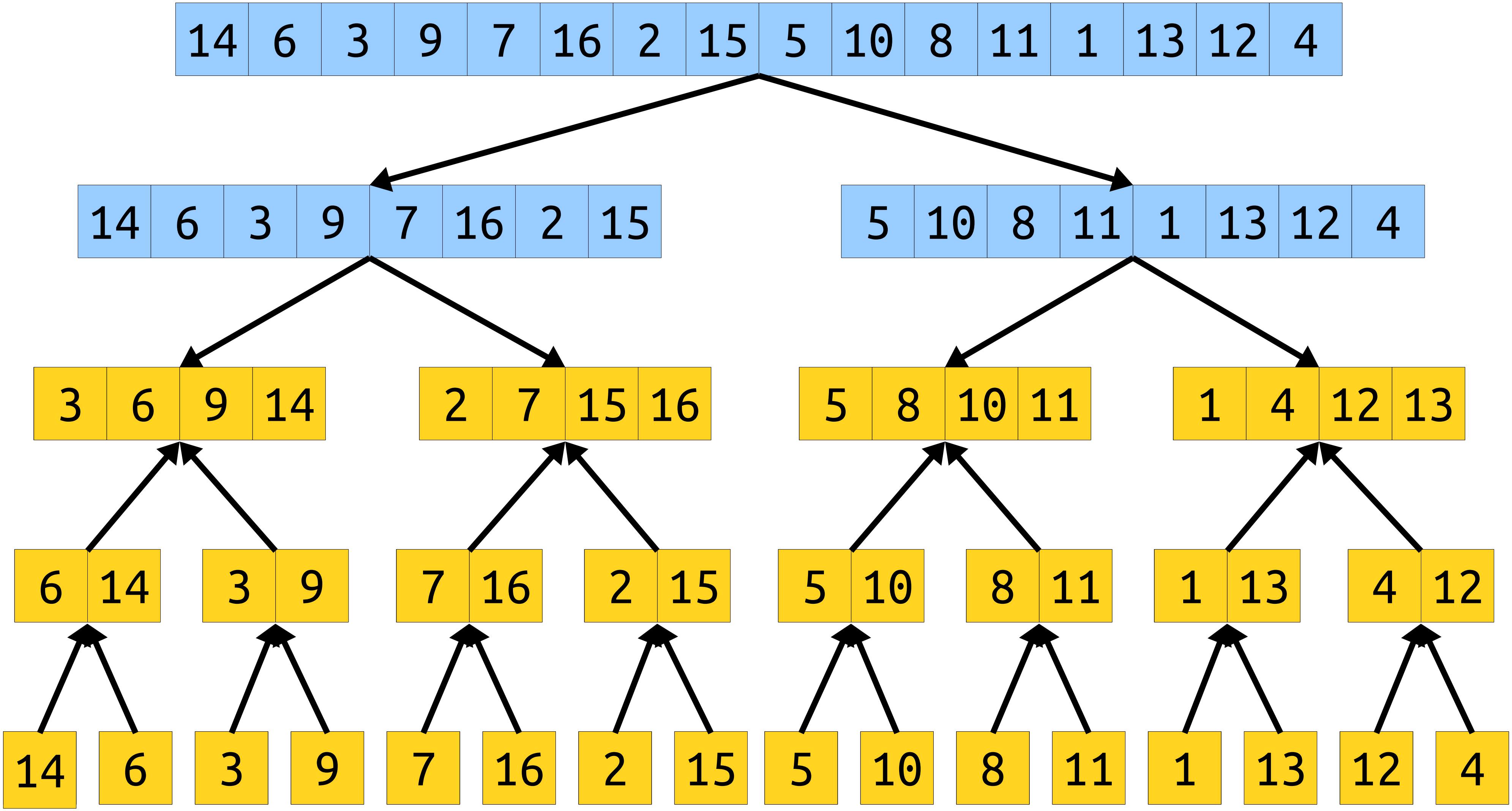


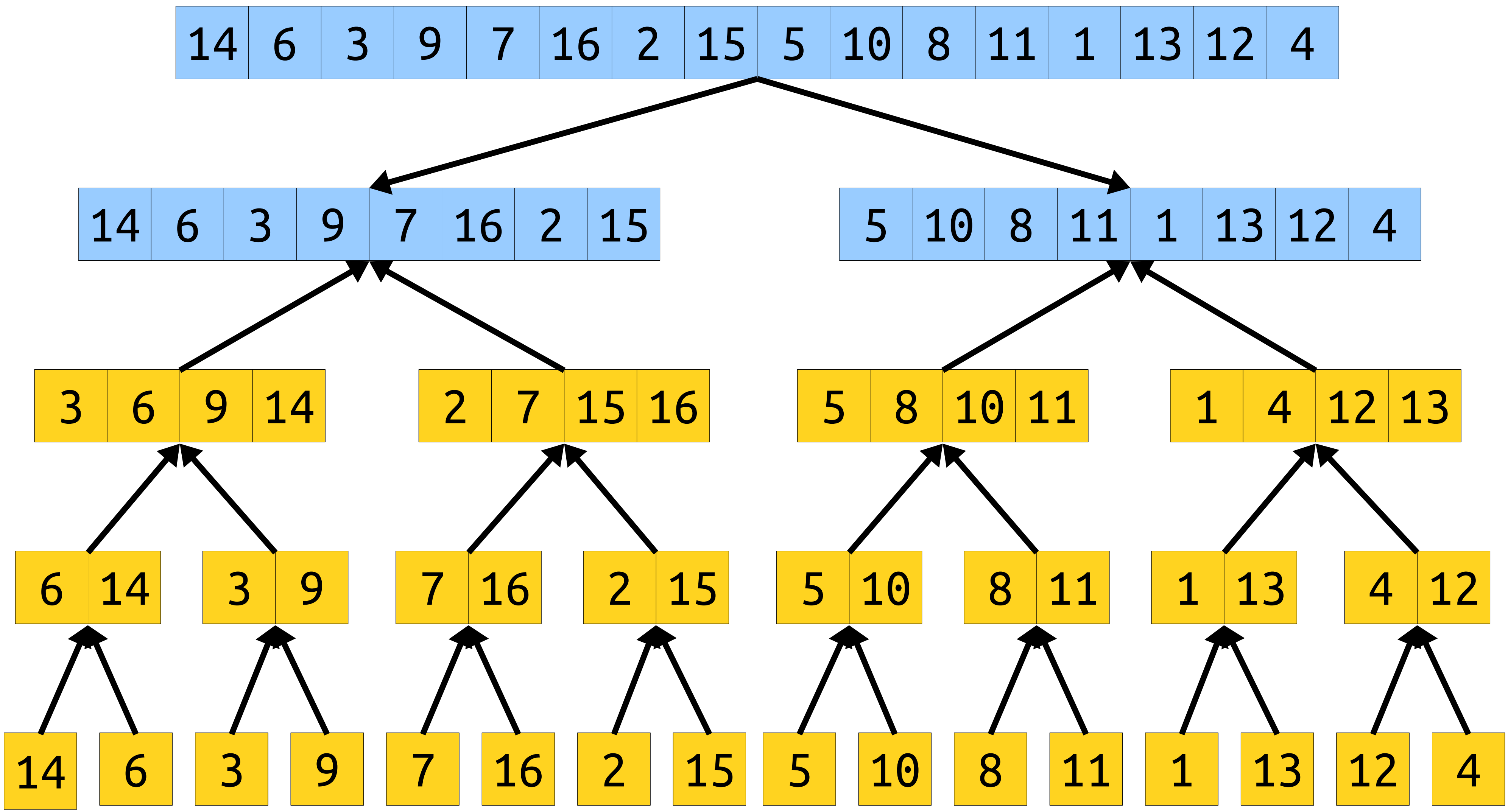
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4



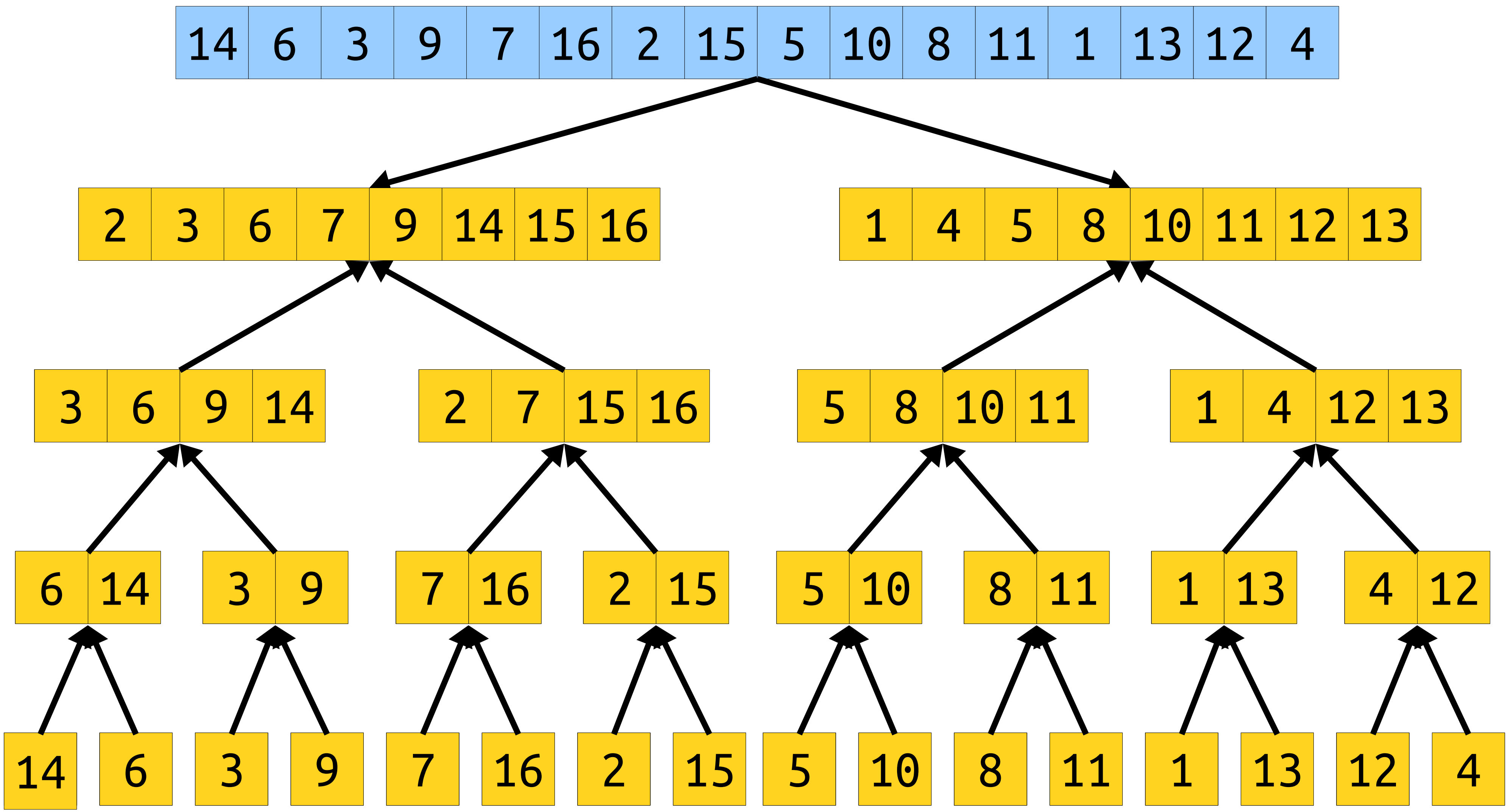
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

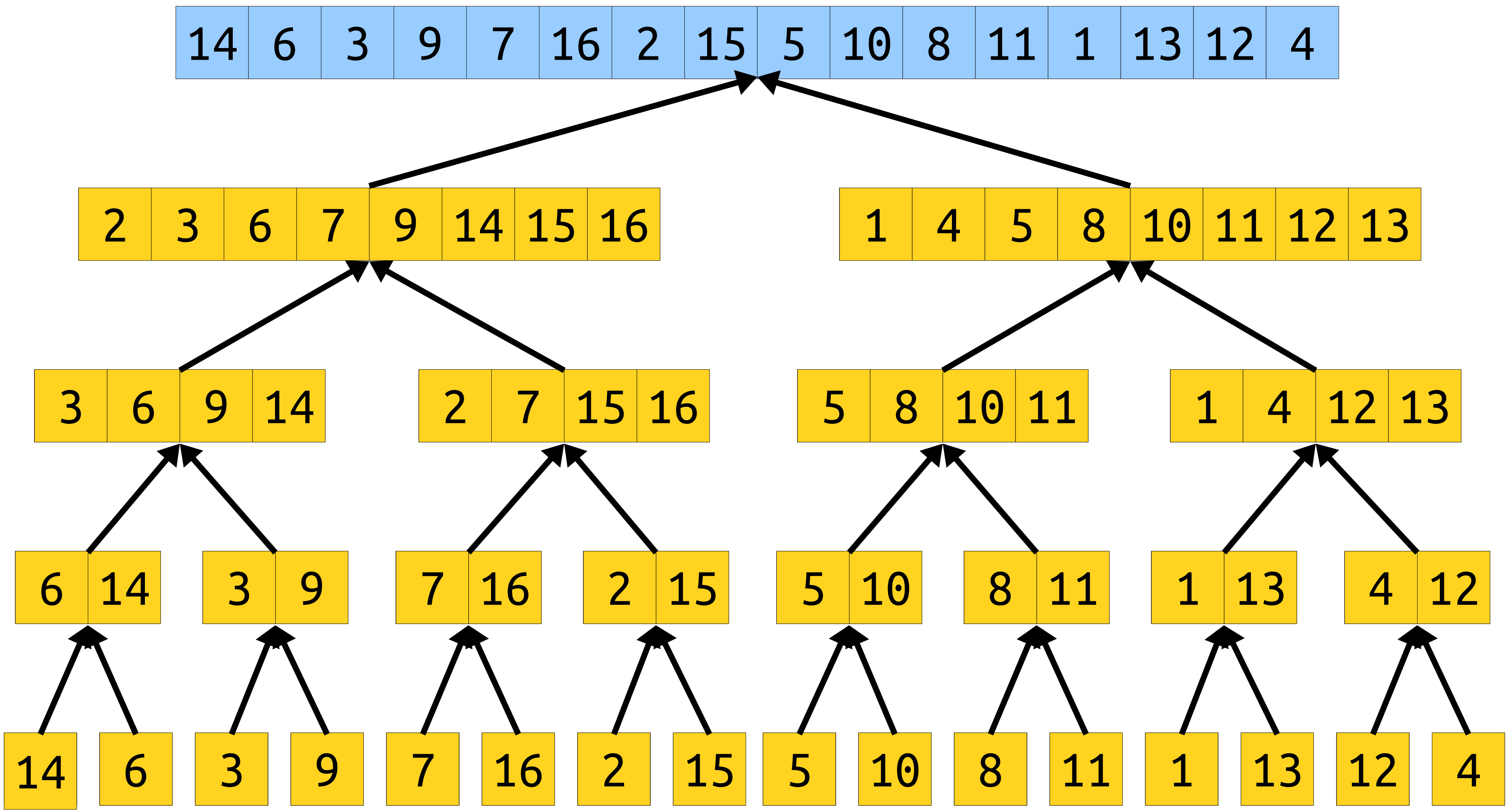

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

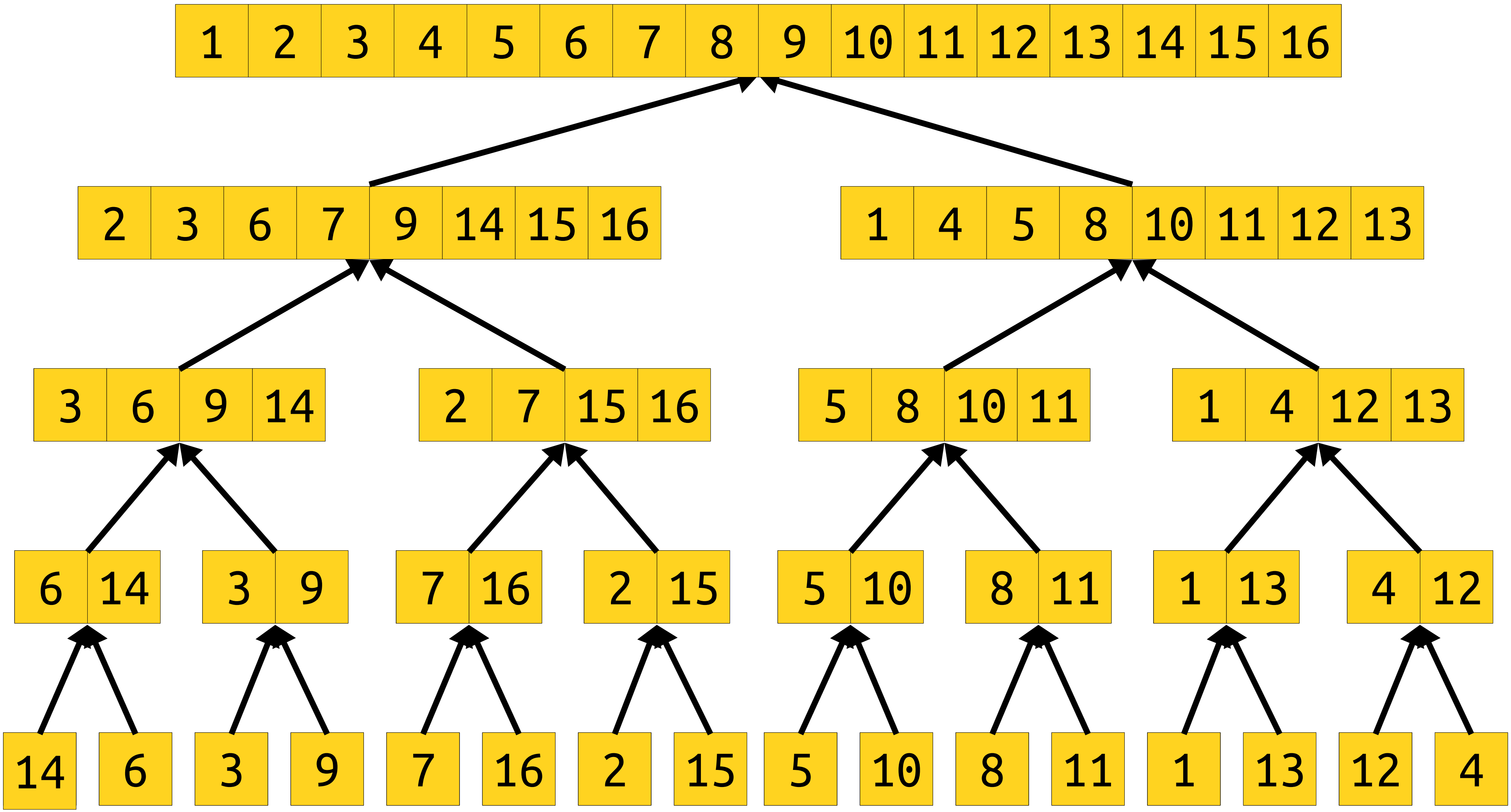

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

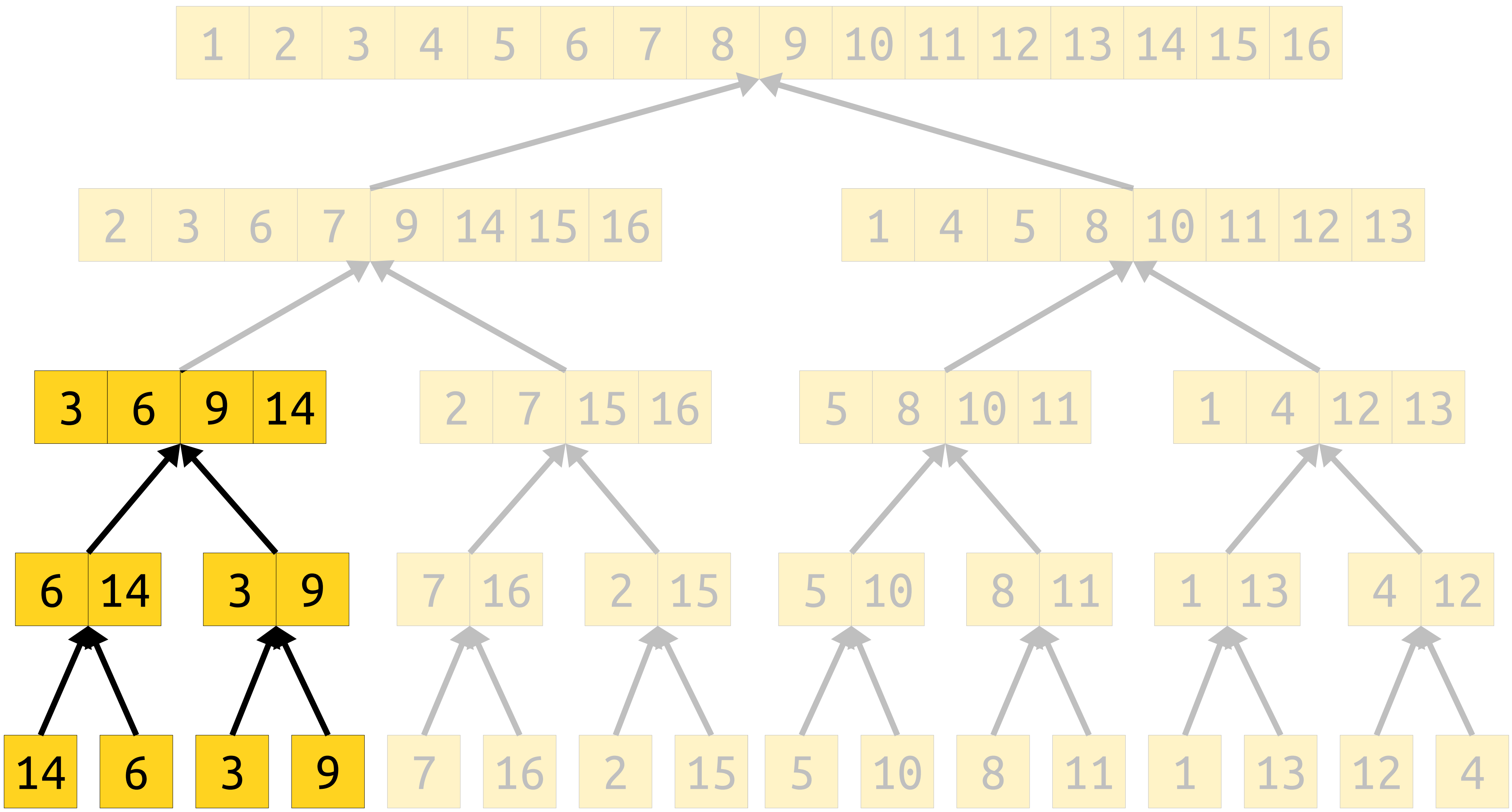


14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

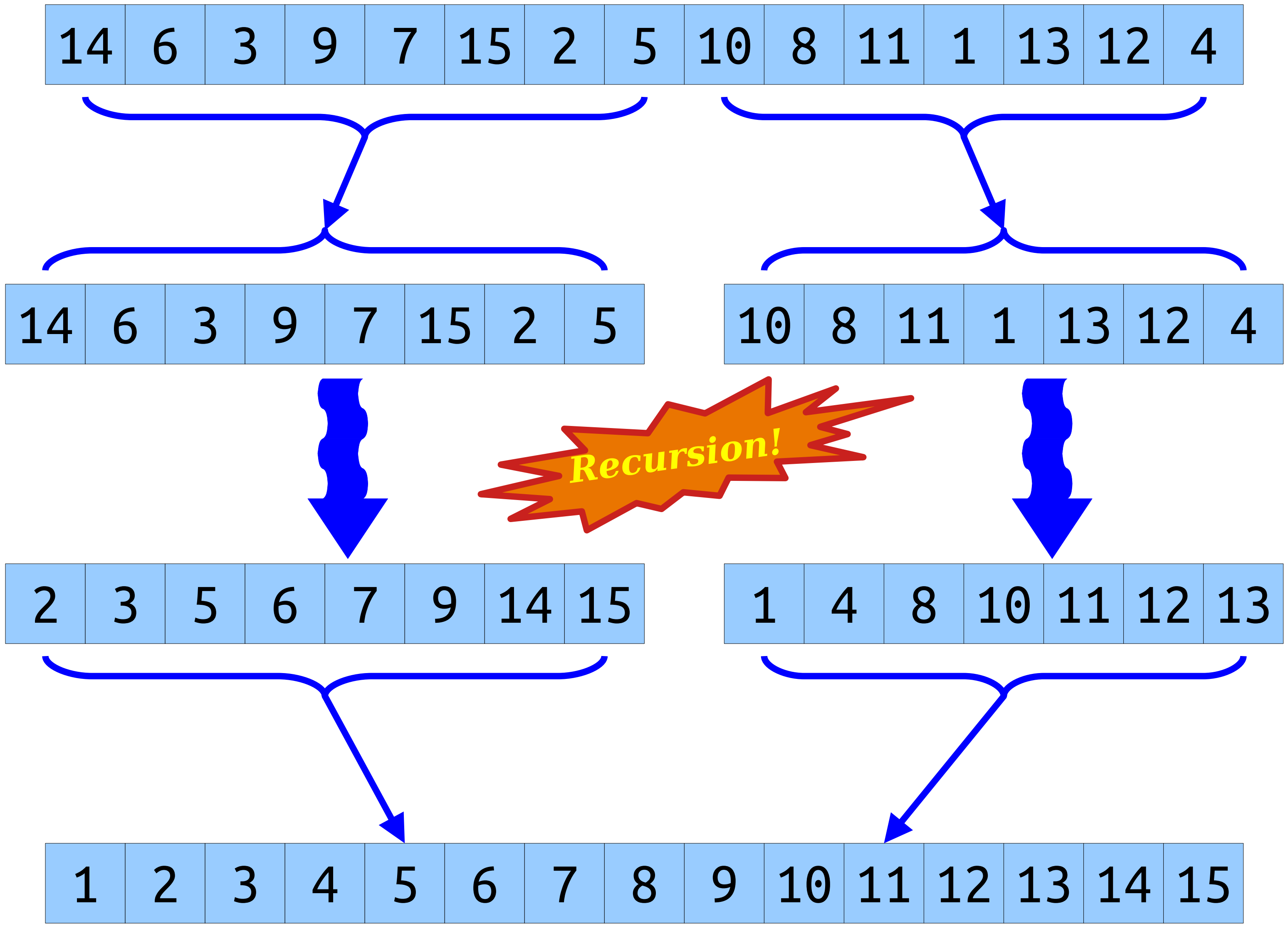

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
2 3 6 7 9 14 15 16
1 4 5 8 10 11 12 13
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4


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


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


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

Mergesort, Intuitively



Split array into roughly equal halves

Recursively mergesort each half

Merge sorted subarrays

Mergsort

- A recursive sorting algorithm!
- ***Base Case:***
  - An empty or single-element list is already sorted.
- ***Recursive step:***
  - Break the list in half and recursively sort each part.
  - 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. */
    int half = v.size() / 2;
    Vector<int> left  = v.subList(0, half);
    Vector<int> right = v.subList(half);

    /* 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!

This next section is the mathiest math we’re going to math all quarter.

It’s great if you can follow along with it.

You aren’t expected to come up with this on your own.

If you like this analysis, take CS161!

```
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. */
   int half = v.size() / 2;
   Vector<int> left  = v.subList(0, half);
   Vector<int> right = v.subList(half);

   /* 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. */
   int half = v.size() / 2;
   Vector<int> left  = v.subList(0, half);
   Vector<int> right = v.subList(half);
```

$O(n)$ work

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

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

$O(n)$ work

```
}
```

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

    /* Split v into two subvectors. */
    int half = v.size() / 2;
    Vector<int> left  = v.subList(0, half);
    Vector<int> right = v.subList(half);

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

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

O(*n*) work

O(*n*) work

Why does forming these sublists take time O(*n*)?

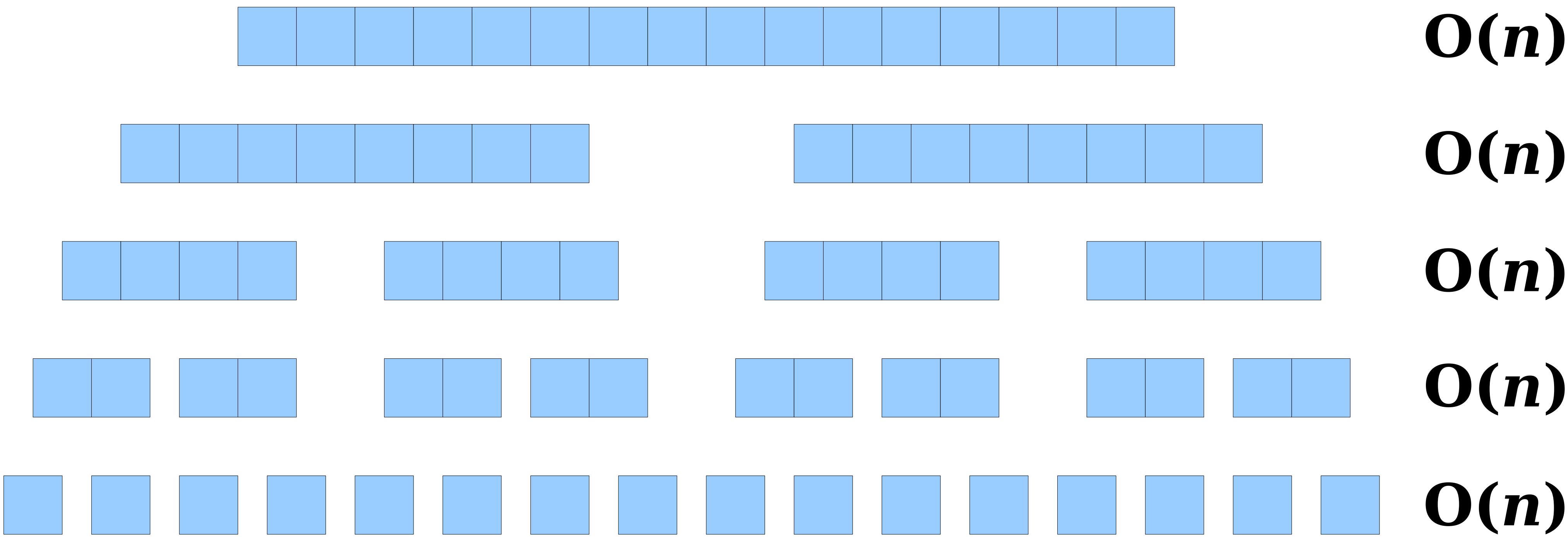
Answer at
https://pollev.com/cs106bwin23

```
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. */
   int half = v.size() / 2;
   Vector<int> left  = v.subList(0, half);
   Vector<int> right = v.subList(half);

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

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

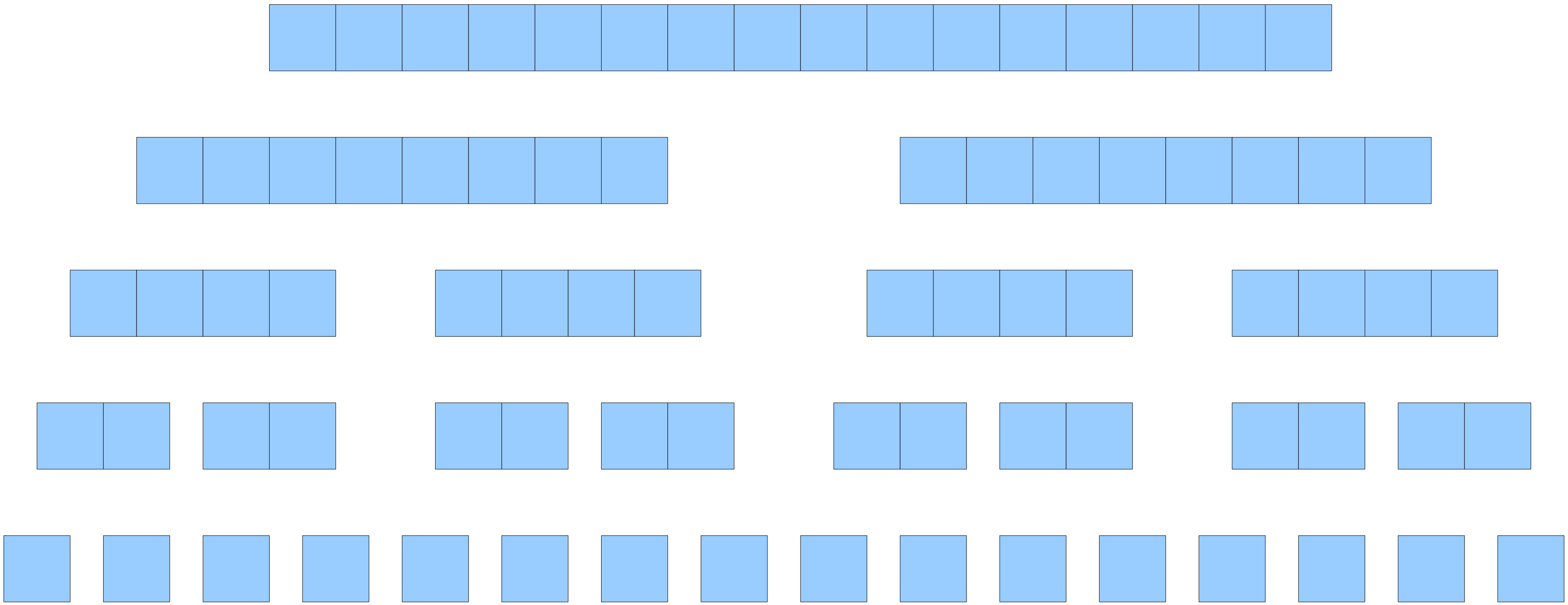
$O(n)$ work

$O(n)$ 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. */
   int half = v.size() / 2;
   Vector<int> left  = v.subList(0, half);
   Vector<int> right = v.subList(half);

   /* 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)

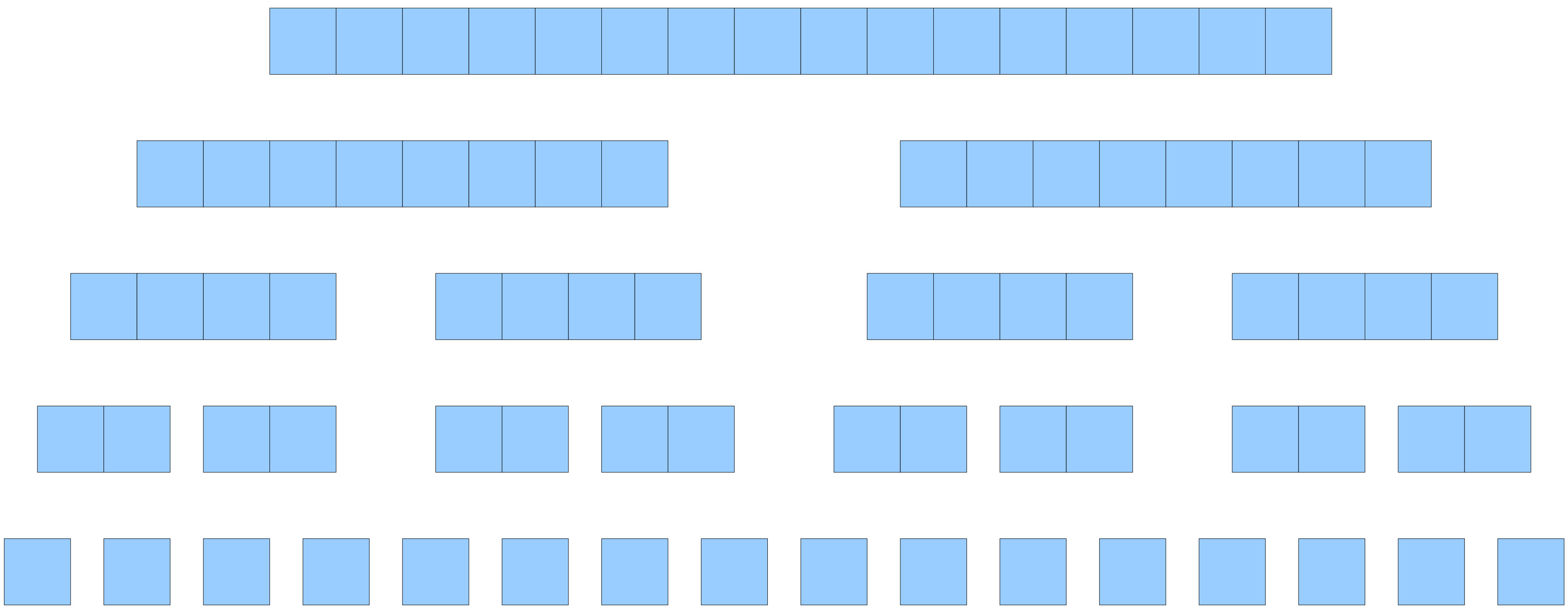
O(n)

O(n)

O(n)

O(n)

How many levels are there?



$O(n)$

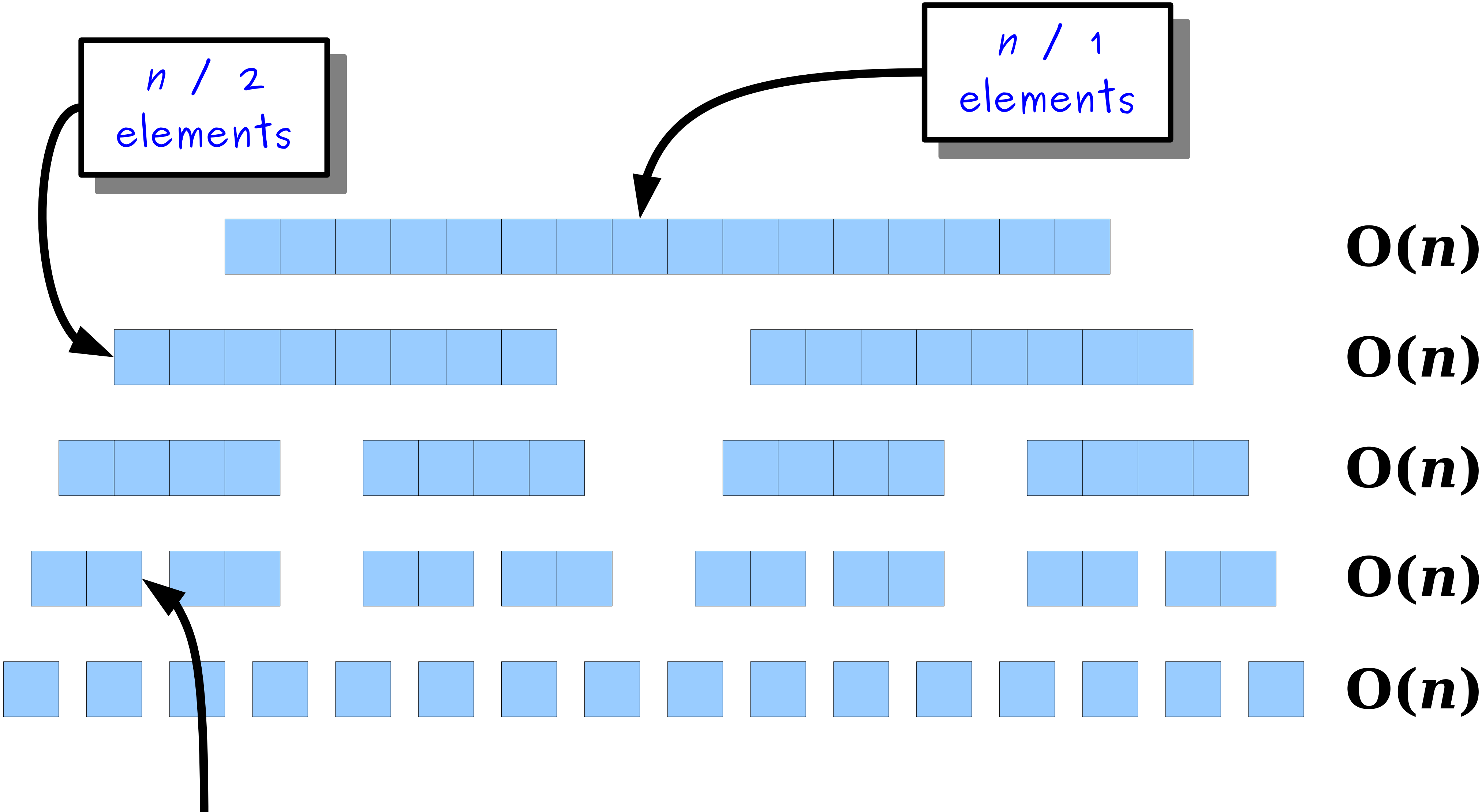
$O(n)$

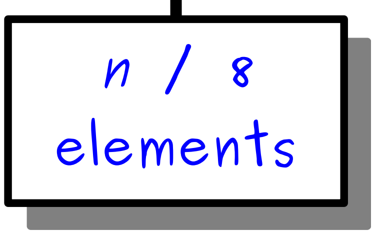
$O(n)$

$O(n)$

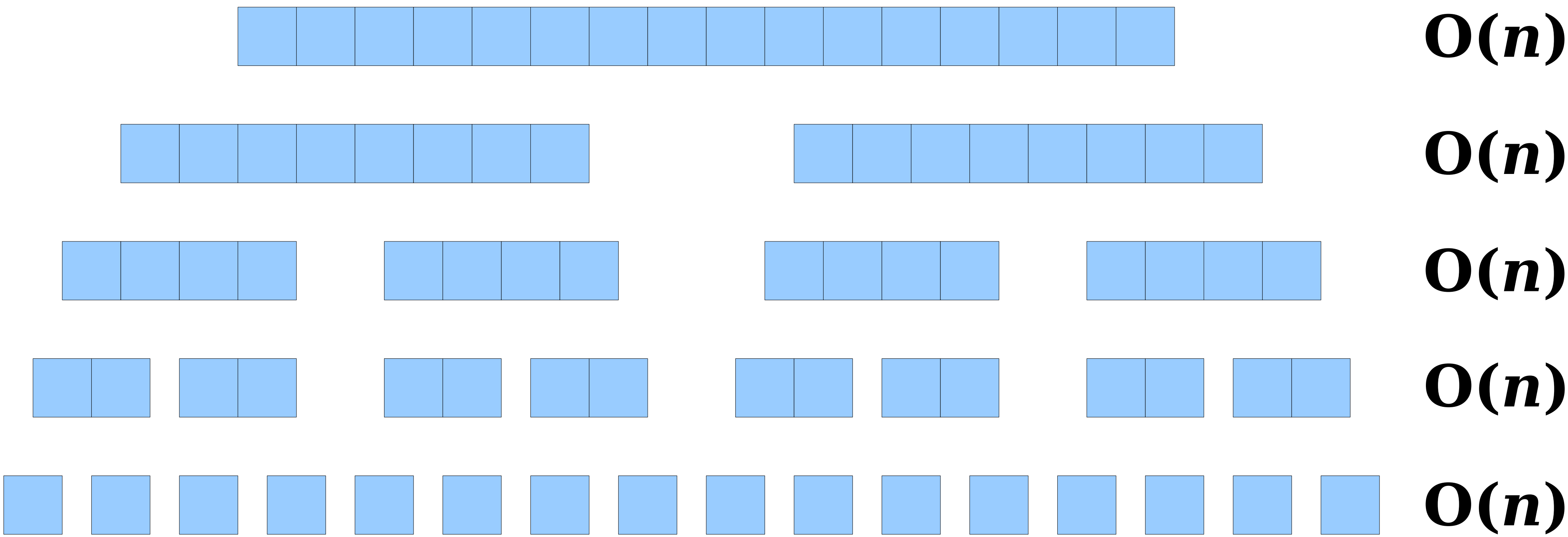
$O(n)$

Each recursive call cuts the
array size in half.

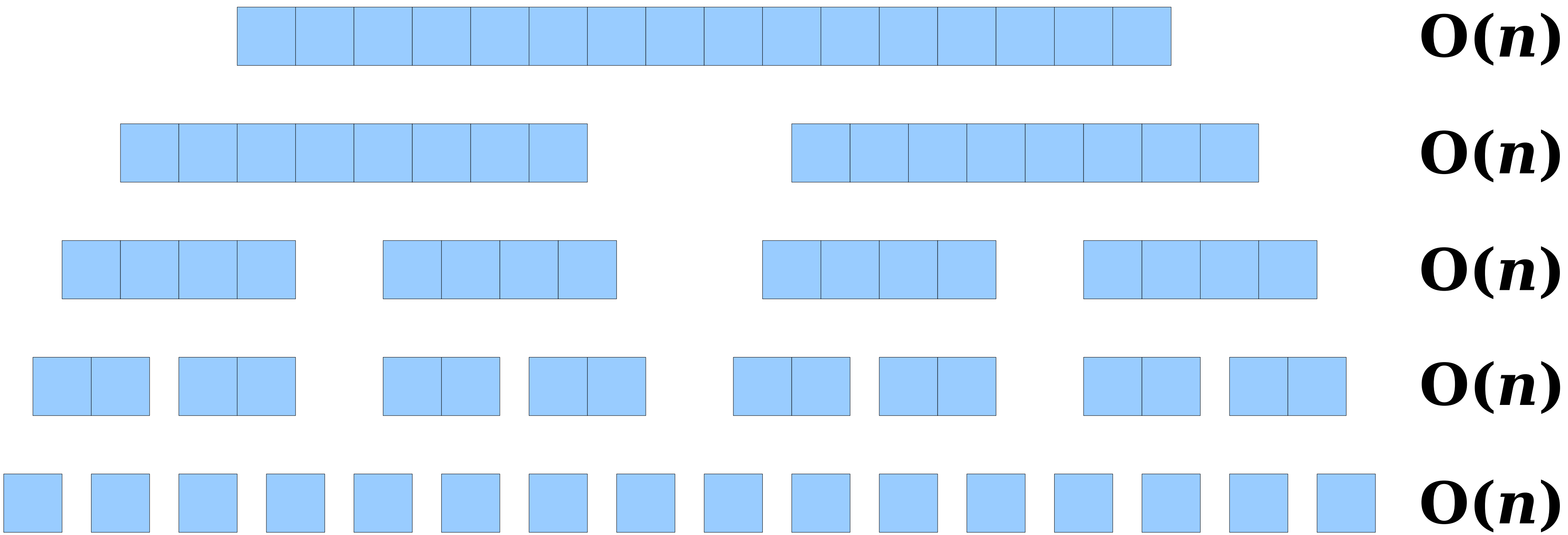




After k layers of the recursion, if the original array has size n, each subarray has size $n / 2^k$.



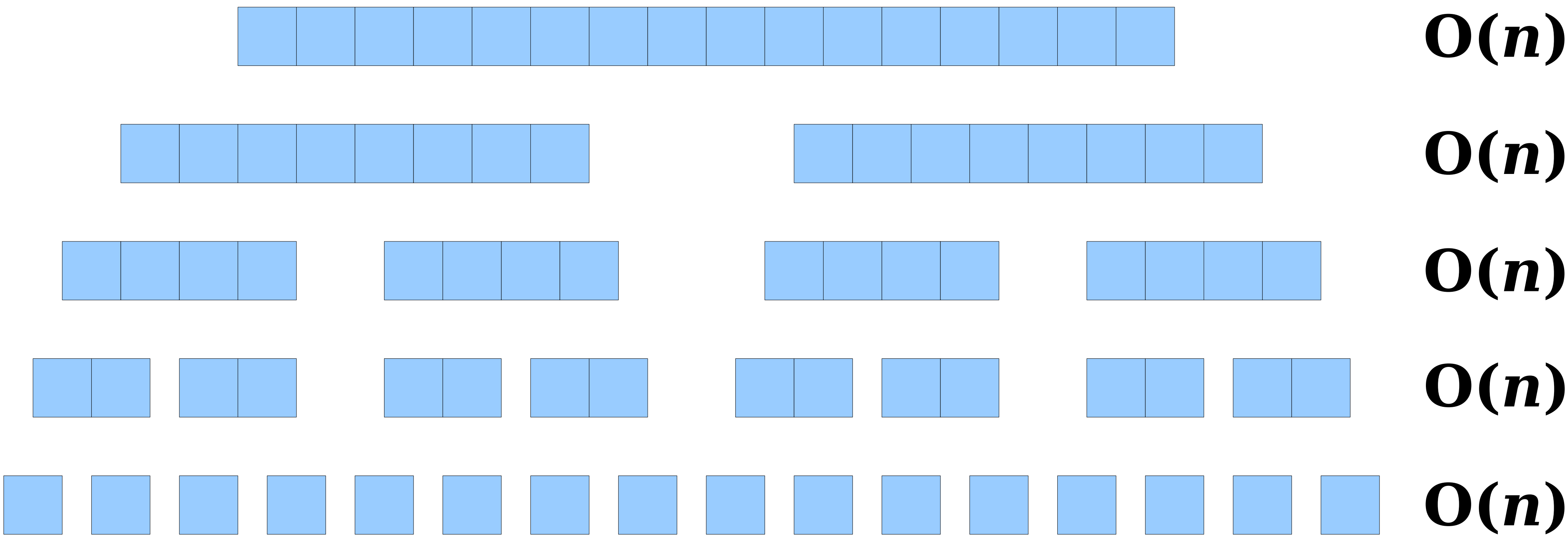
The recursion stops when we're down to a single element.



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: $k = \mathbf{\log_2 n}$.



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?
- A ***comparison sort*** is a sorting algorithm that only learns the relative ordering of its elements by making comparisons between elements.
  - All of the sorting algorithms we've seen so far are comparison sorts.
- ***Theorem:*** There are no comparison sorts whose average-case runtime can be better than O(n log n).
- If we stick with making comparisons, we can only hope to improve on mergesort by a constant factor!

A Quick Historical Aside

- Mergesort was one of the first algorithms developed for computers as we know them today.
- It was invented by John von Neumann in 1945 (!) as a way of validating the design of the first “modern” (stored-program) computer.
- Want to learn more about what he did? Check out ***this article*** by Stanford’s very own Donald Knuth.

Time-Out for Announcements!

Midterm Review Session

- The amazing SL team will be holding a midterm review session this weekend:

Saturday, February 11th

3:30 – 6:00PM

Hewlett 200

- There's an ***online poll*** where you can vote on what you'd like the team to cover.

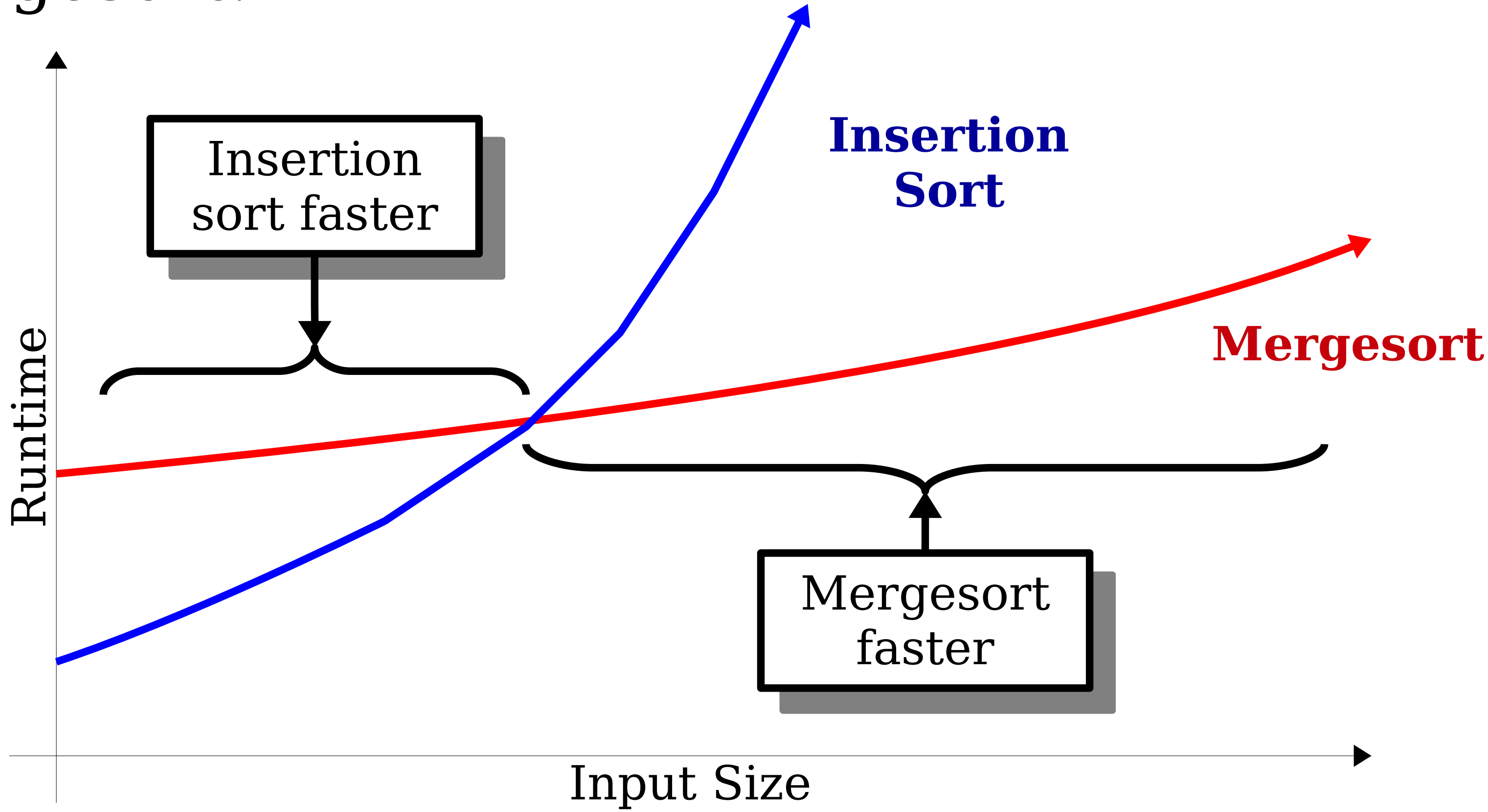
```
lecture.notify_all();
```

(A C++ command to wake up parts of the program that are sleeping and waiting for a signal to continue.)

Improving Mergesort

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 {
        int half = v.size() / 2;
        Vector<int> left  = v.subList(0, half);
        Vector<int> right = v.subList(half);

        hybridMergesort(left);
        hybridMergesort(right);

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

Hybrid Mergesort

```
void hybridMergesort(Vector<int>& v) {
    if (v.size() <= kCutoffSize) {
        insertionSort(v);
    } else {
        int half = v.size() / 2;
        Vector<int> left  = v.subList(0, half);
        Vector<int> right = v.s

        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?

Hybrid Mergesort

```
void hybridMergesort(Vector<int>& v) {
    if (v.size() <= kCutoffSize) {
        insertionSort(v);
    } else {
        int half = v.size() / 2;
        Vector<int> left  = v.subList(0, half);
        Vector<int> right = v.subList(half);

        hybridMergesort(left);
        hybridMergesort(right);

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

Why Sort?

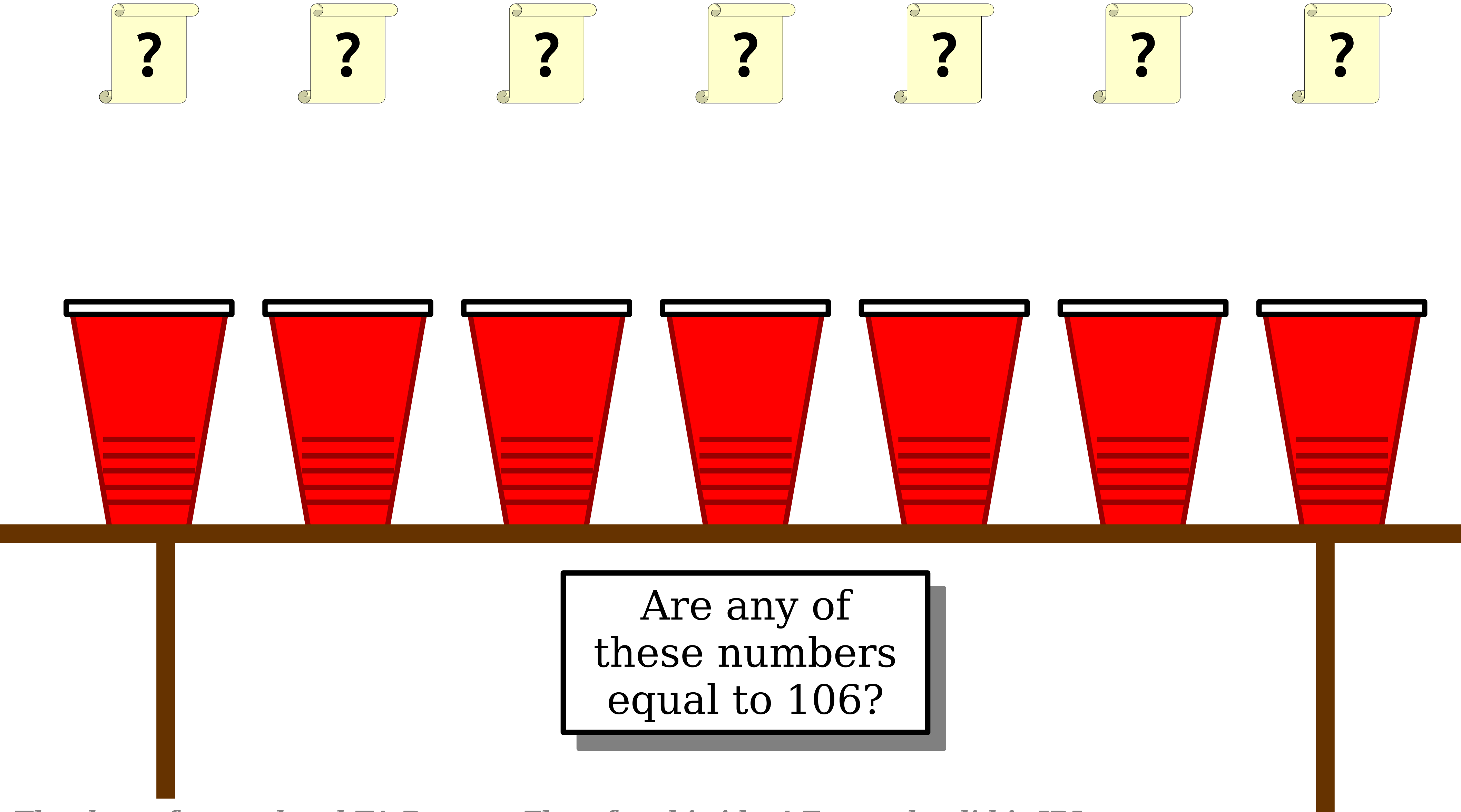
Suppose we want to search an array for an element, and we know that array is sorted.

We could scan from left to right to find that element, but that takes time $O(n)$.

Can we take advantage of the fact that the list is sorted?

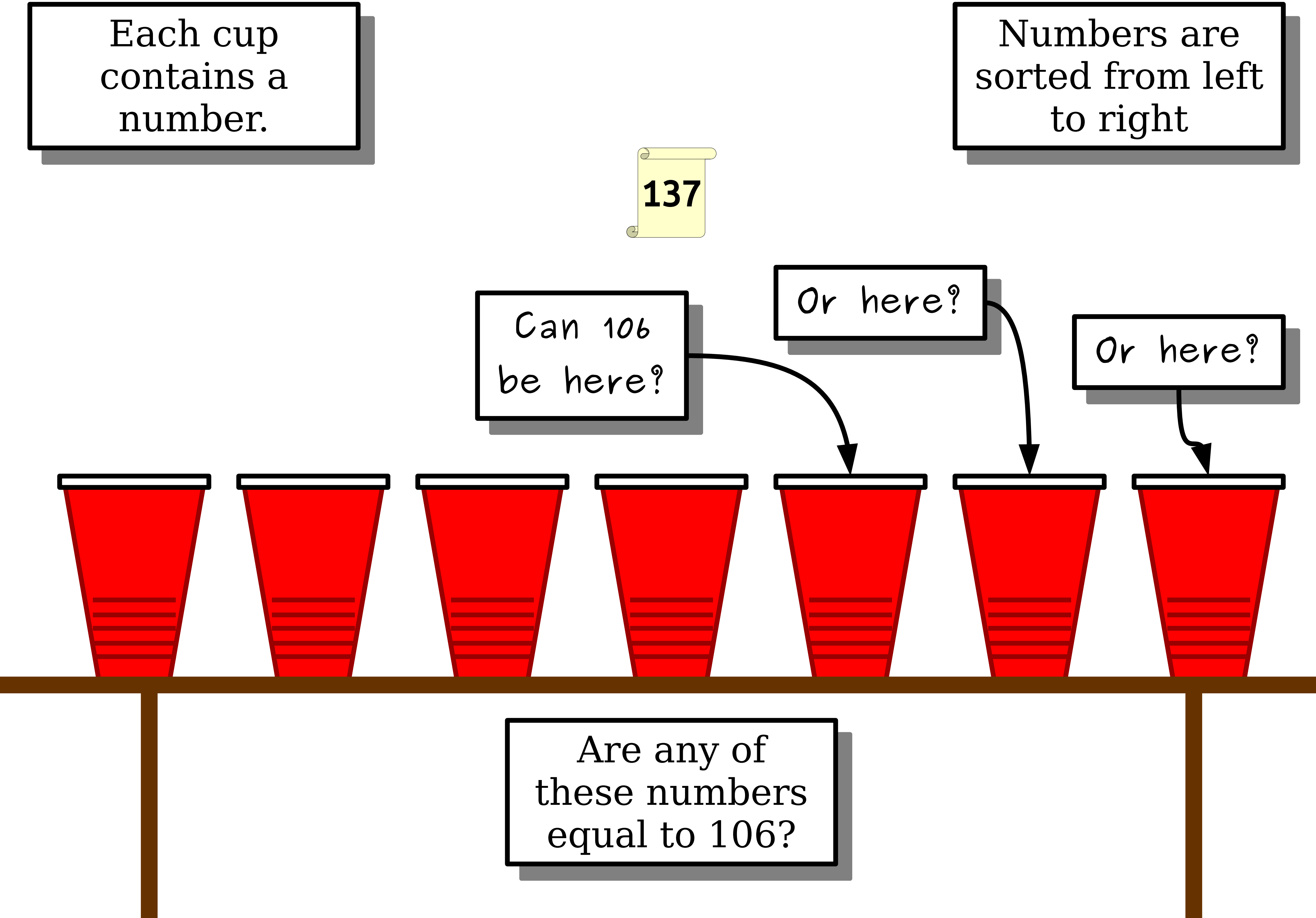
Each cup contains a number.

Numbers are sorted from left to right



Are any of these numbers equal to 106?

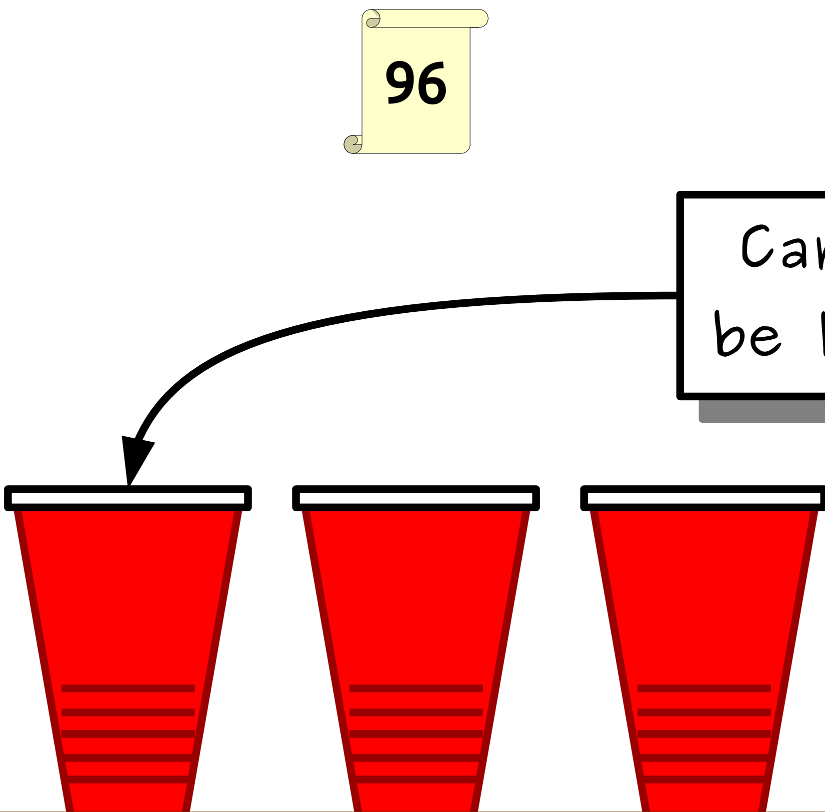
Thanks to former head TA Dawson Zhou for this idea! Except he did it IRL.



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

Each cup contains a number.

Numbers are sorted from left to right



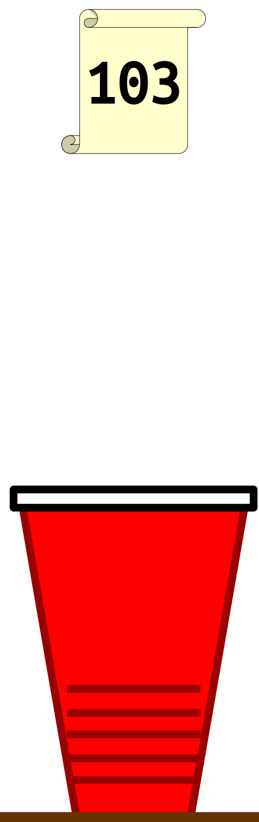
Can 106 be here?

Are any of these numbers equal to 106?

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

Each cup contains a number.

Numbers are sorted from left to right



Are any of these numbers equal to 106?

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

Each cup contains a number.

Numbers are sorted from left to right

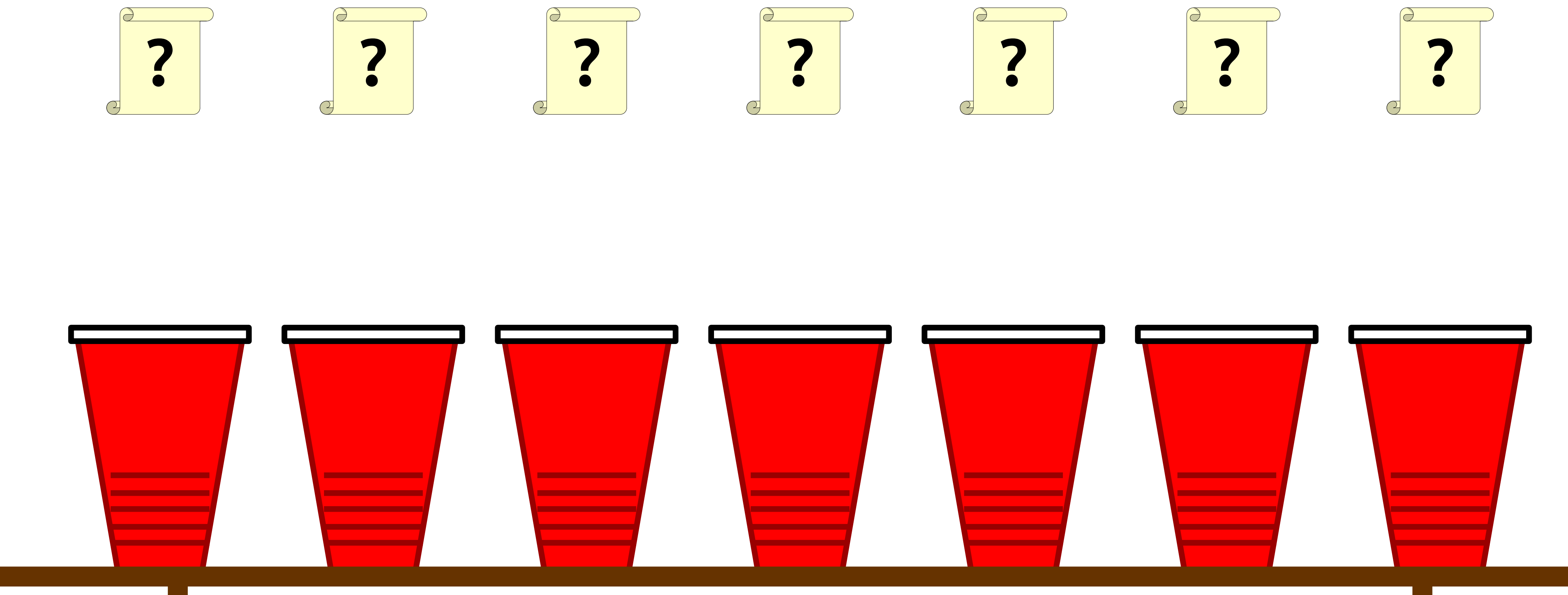
Alas, 106 is not to be found here.

Are any of these numbers equal to 106?

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

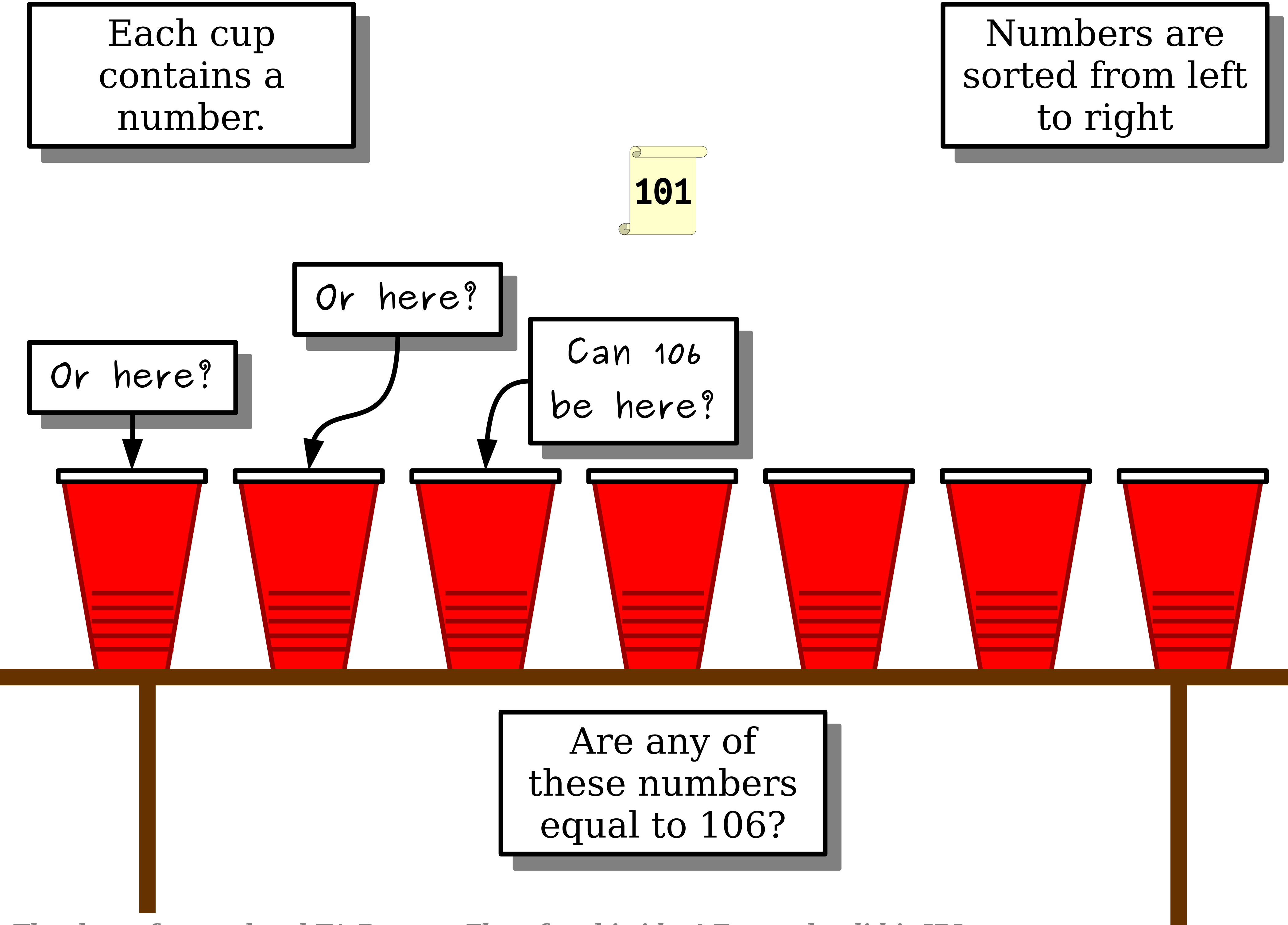
Each cup contains a number.

Numbers are sorted from left to right

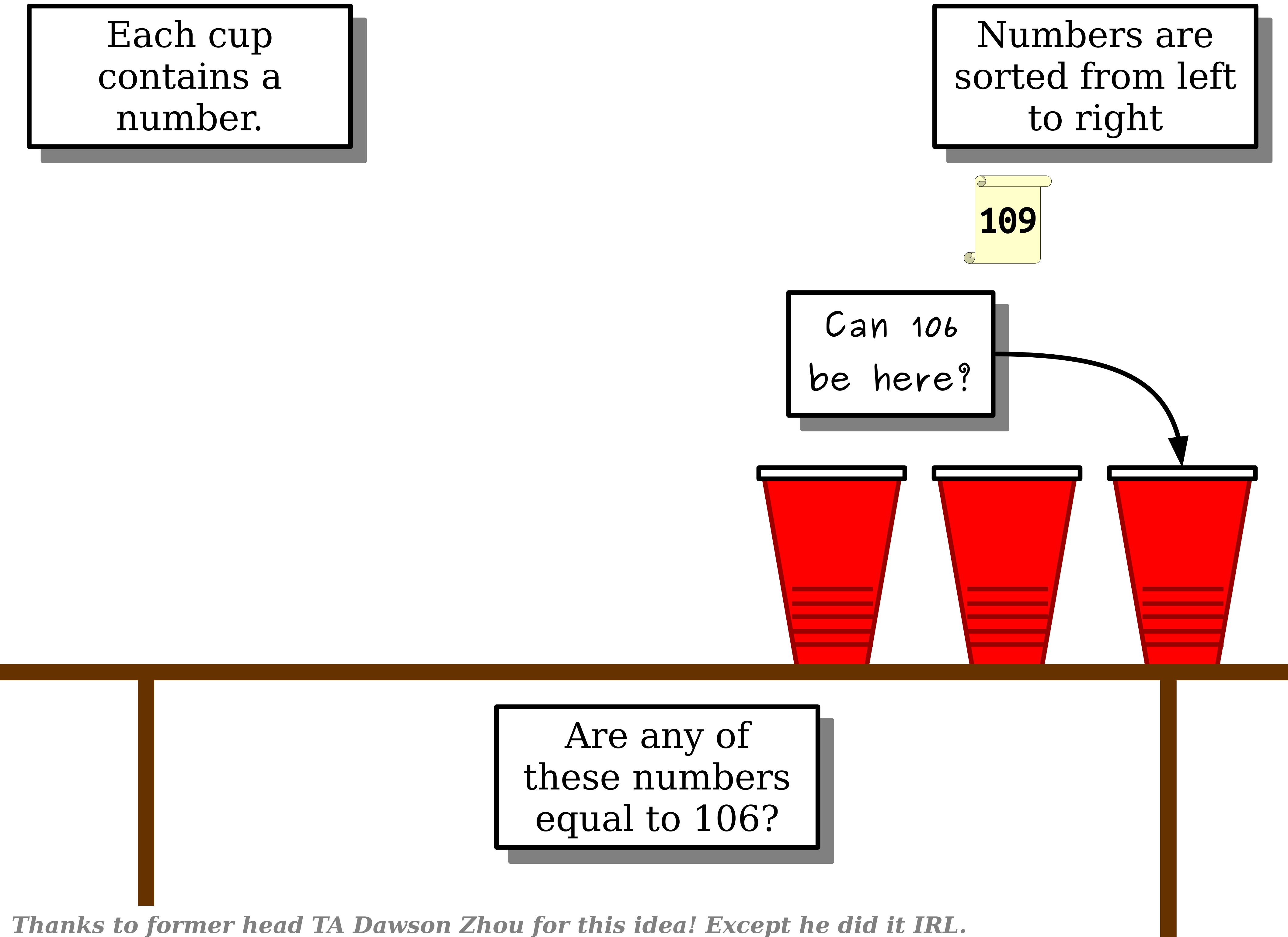


Are any of these numbers equal to 106?

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



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


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

Each cup contains a number.

Numbers are sorted from left to right

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);
    }
}

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

Question to ponder:
how does this code correspond to the example from earlier?

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 scanning from the left to the right!

Why All This Matters

- Big-O notation gives us a ***quantitive way*** to predict runtimes.
- Those predictions provide a ***quantitive 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

- ***Read Chapter 10 of the textbook.***
  - It’s all about big-O and sorting.
- ***Finish Assignment 4.***
  - We’re here for you if you need help!
- ***Study for the Midterm***
  - Review old assignments, do practice exams, etc.

Next Time

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