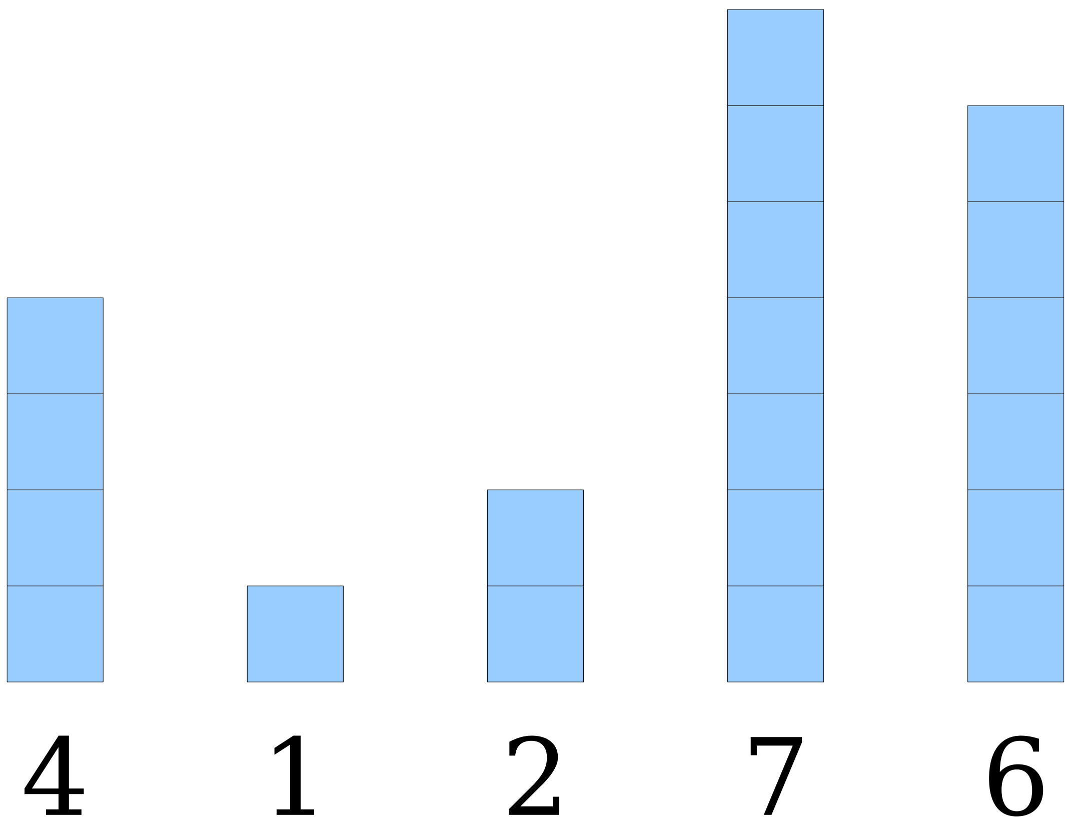
Searching and Sorting

Part Two

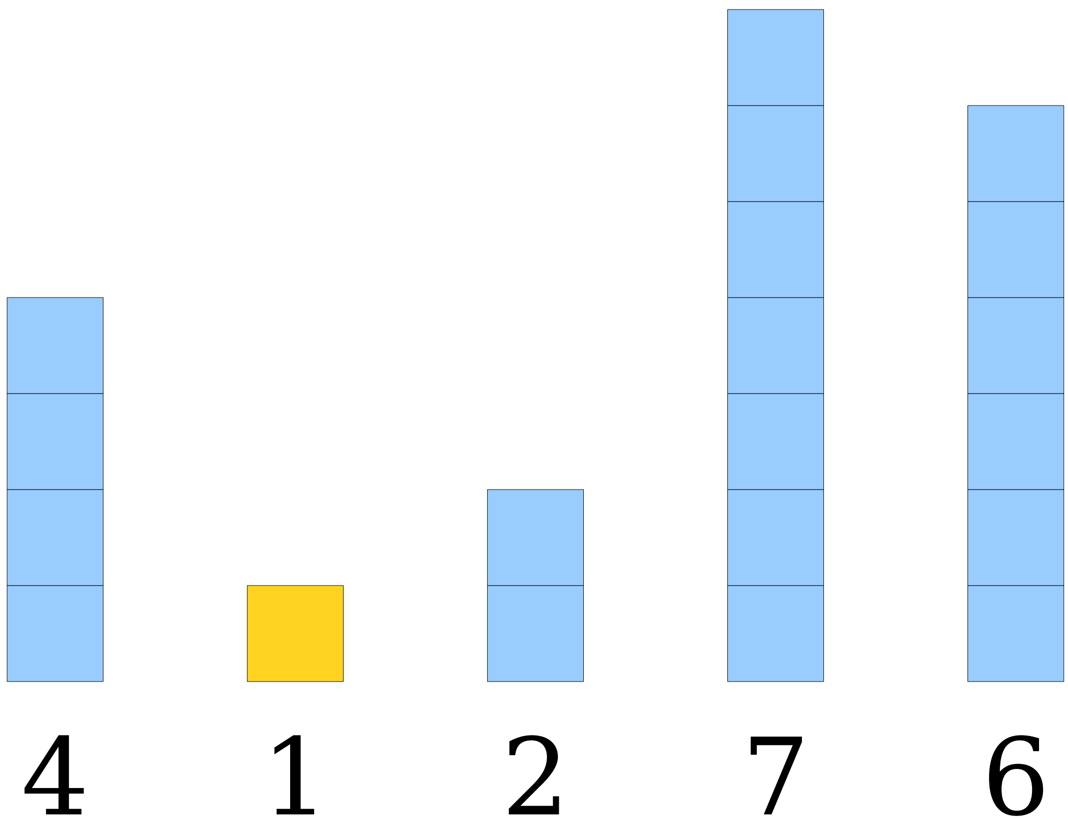
Recap from Last Time

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

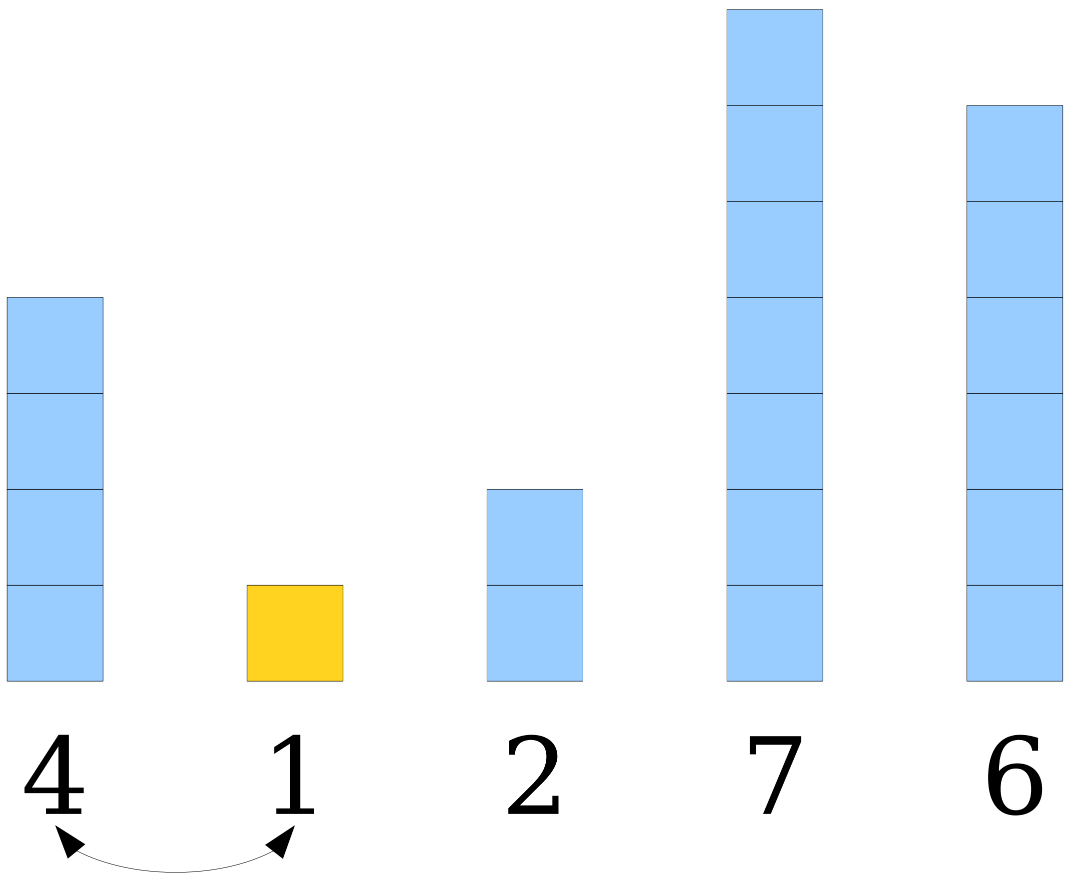
An Initial Idea: ***Selection Sort***



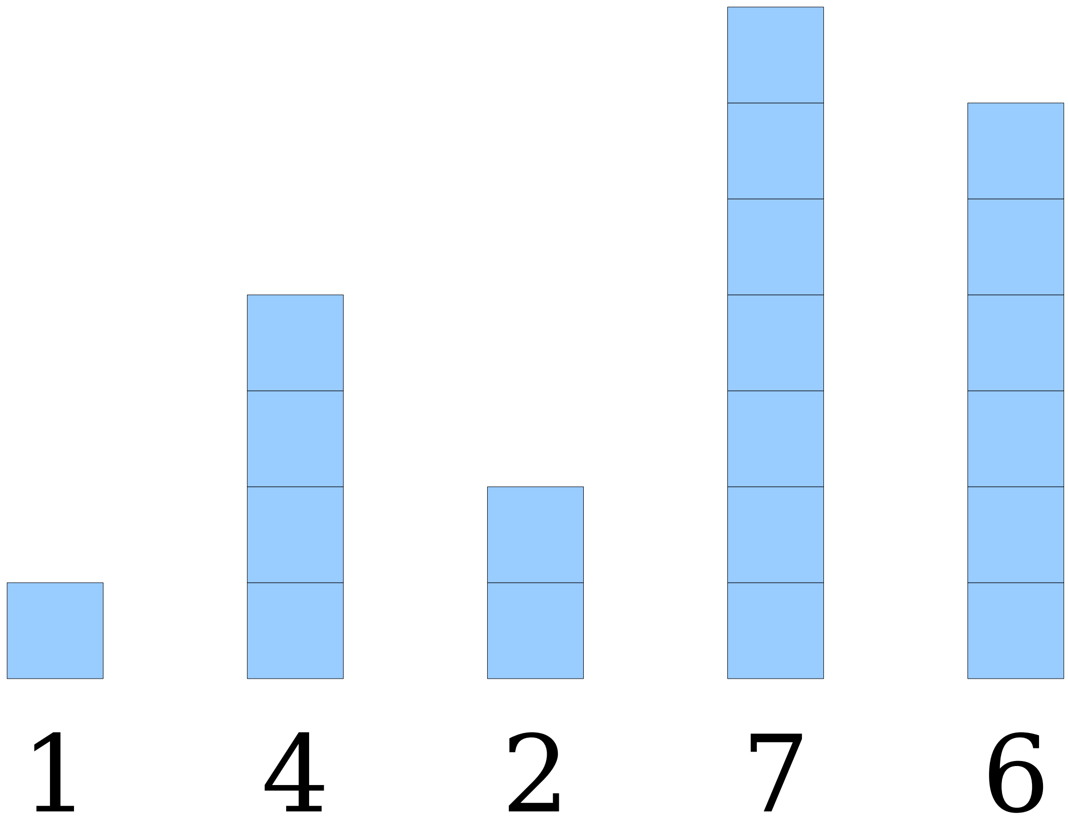
An Initial Idea: ***Selection Sort***



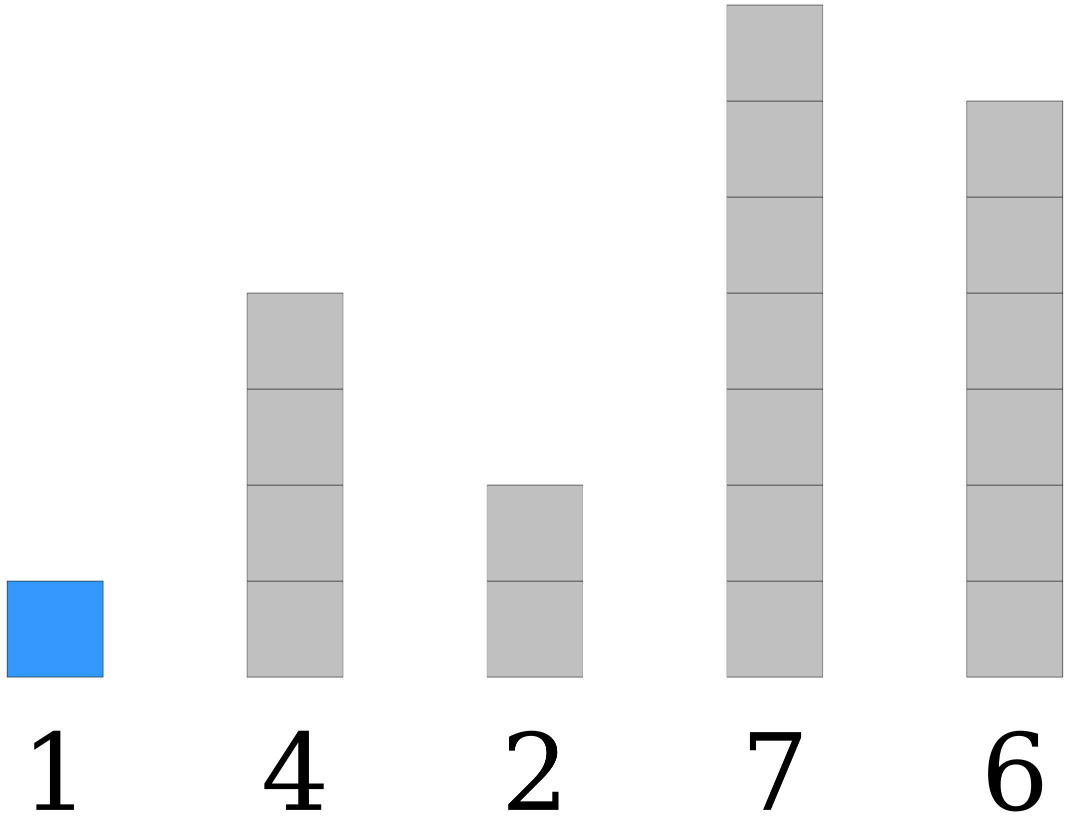
An Initial Idea: ***Selection Sort***



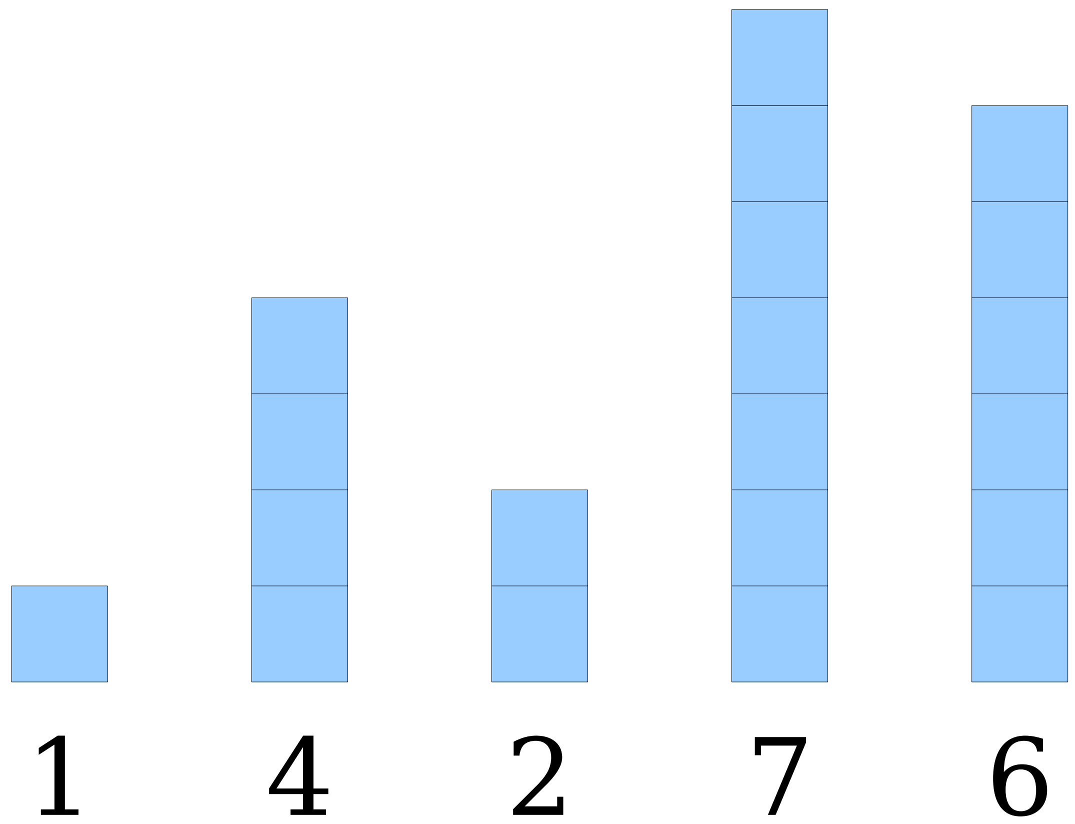
An Initial Idea: ***Selection Sort***



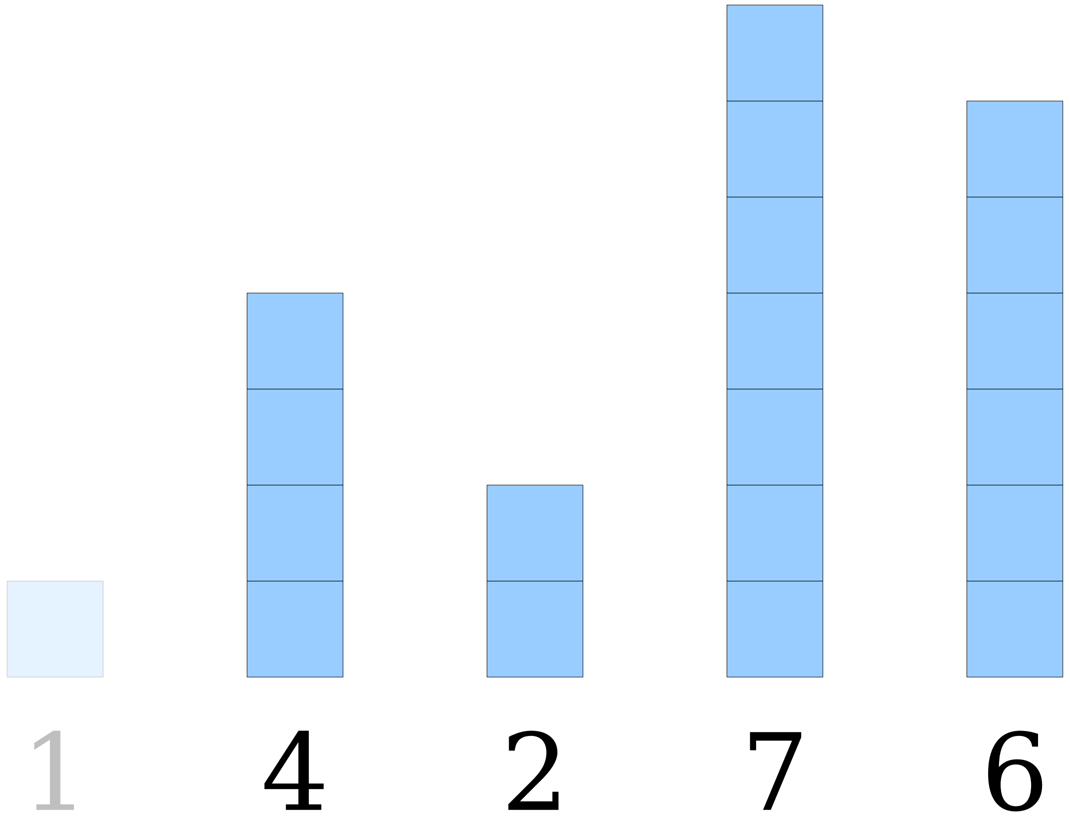
An Initial Idea: ***Selection Sort***



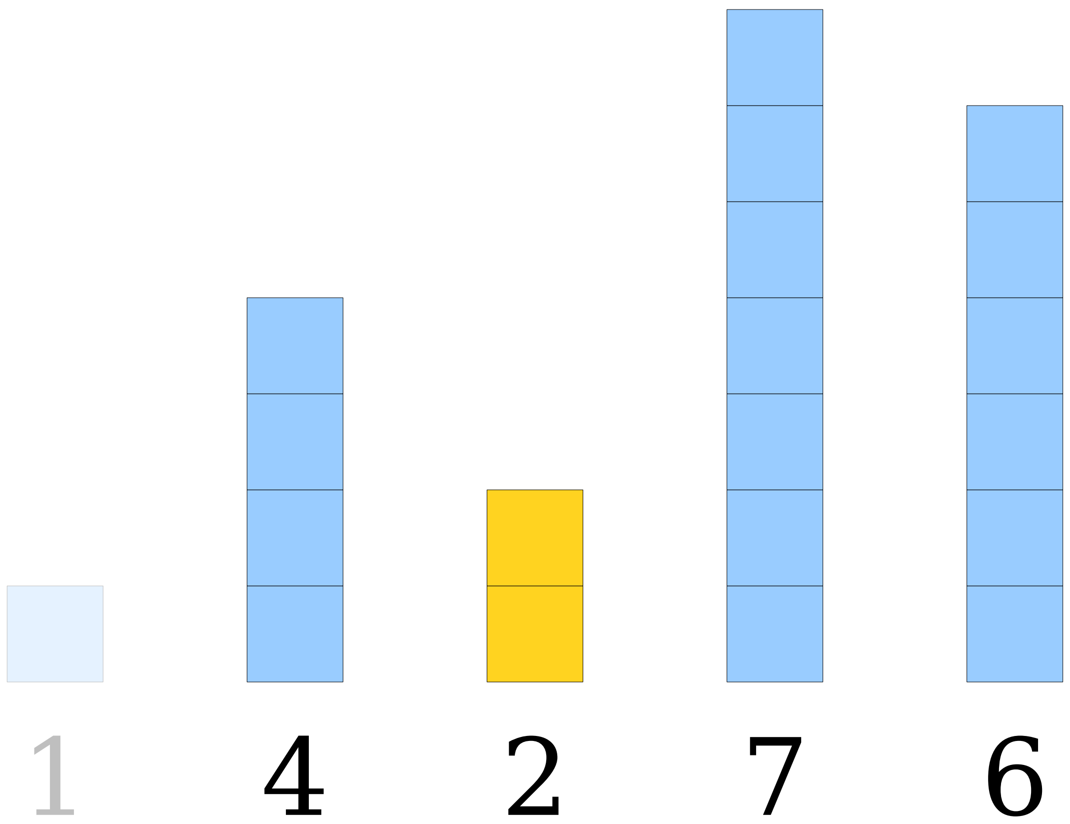
An Initial Idea: ***Selection Sort***



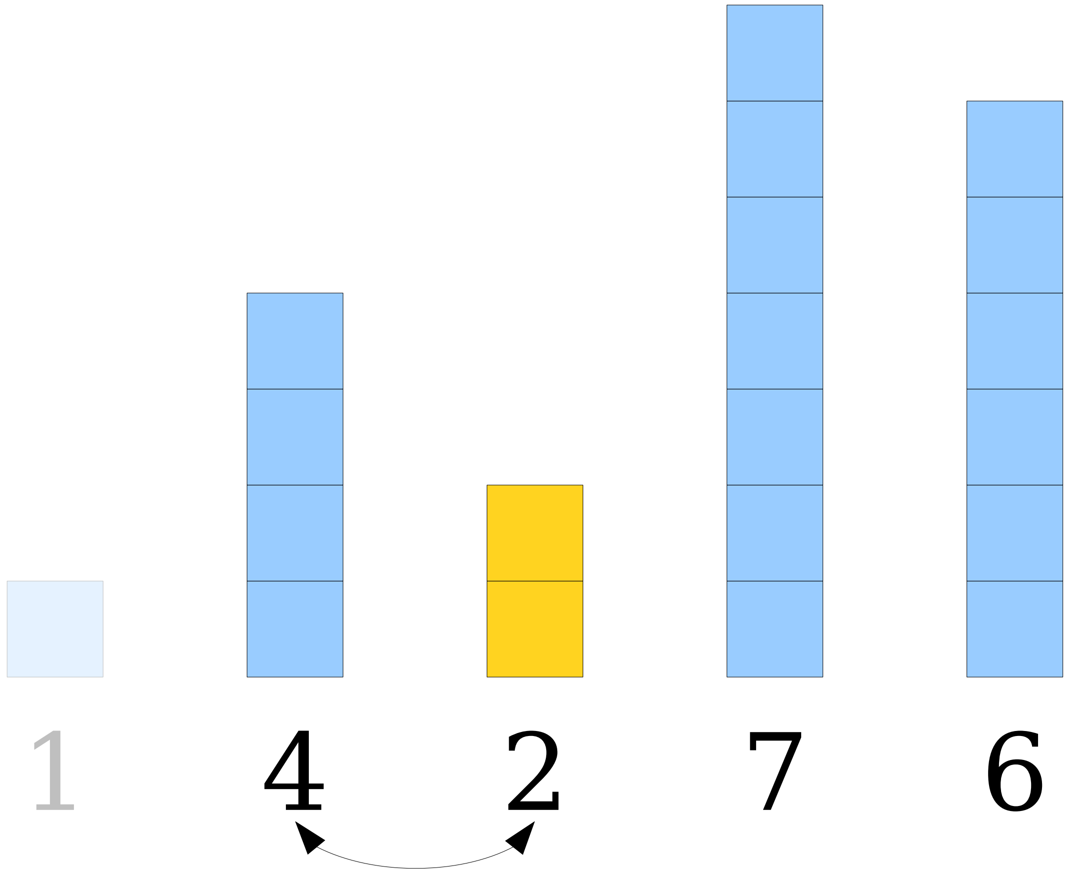
An Initial Idea: ***Selection Sort***



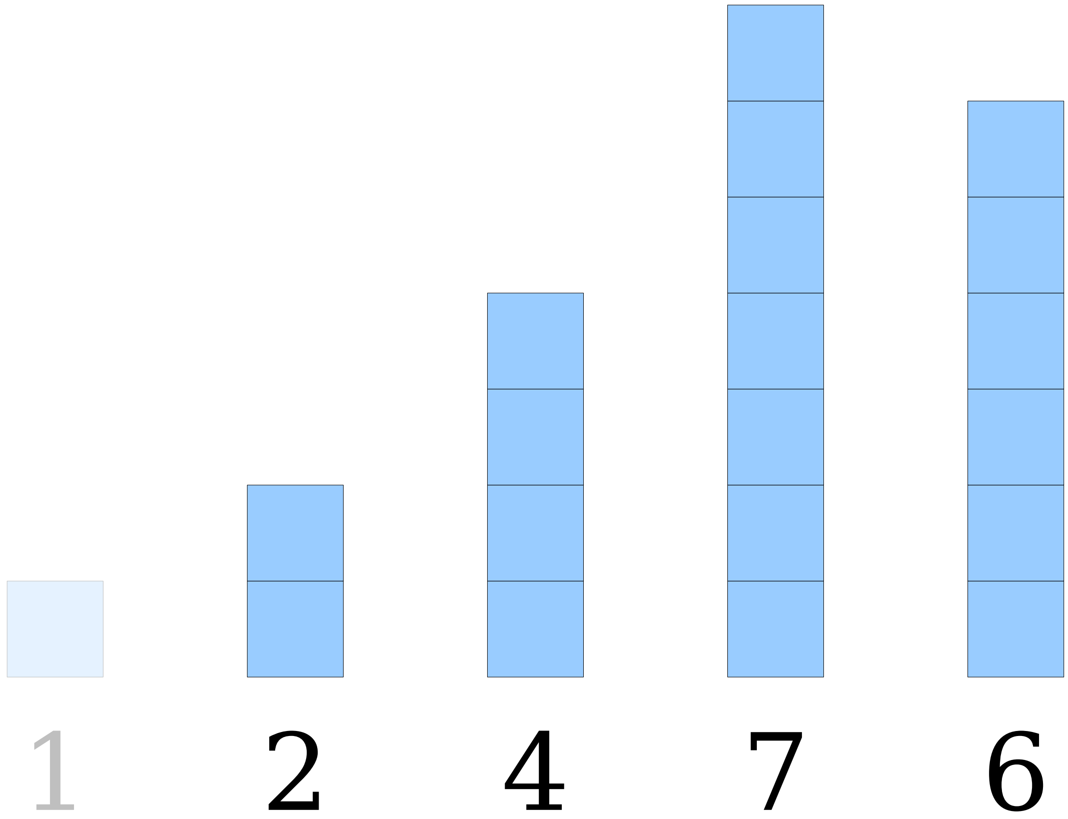
An Initial Idea: ***Selection Sort***



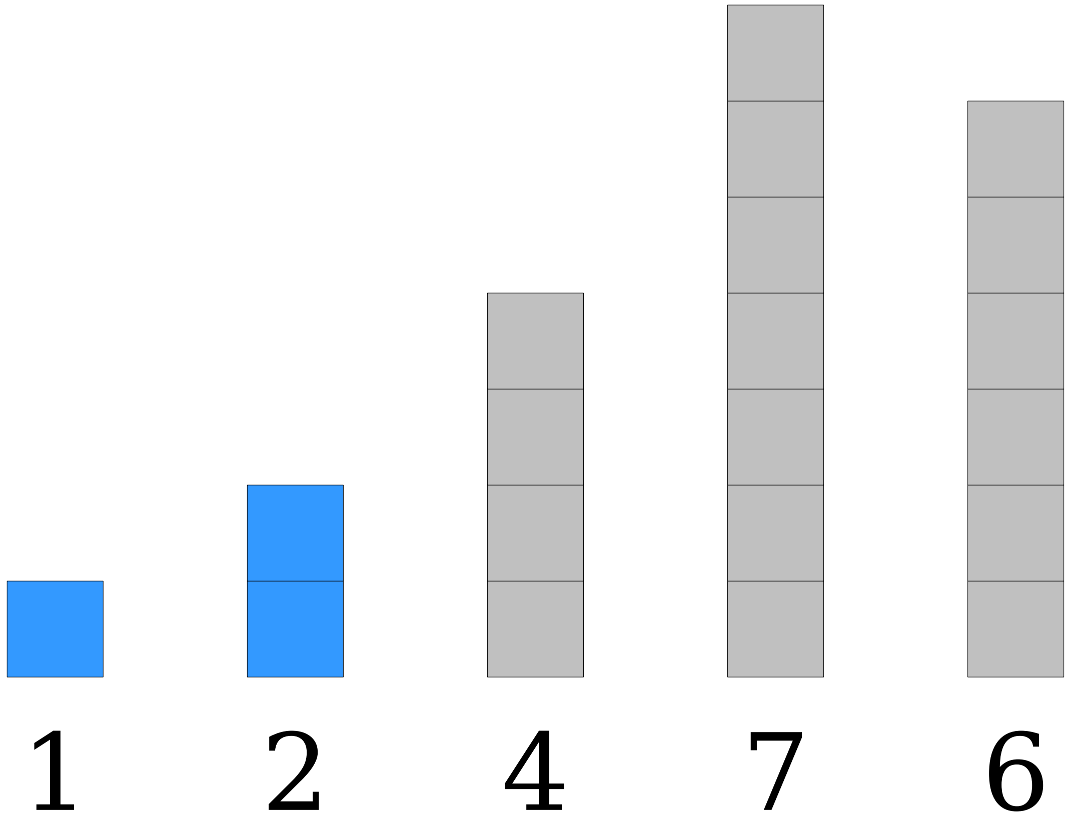
An Initial Idea: ***Selection Sort***



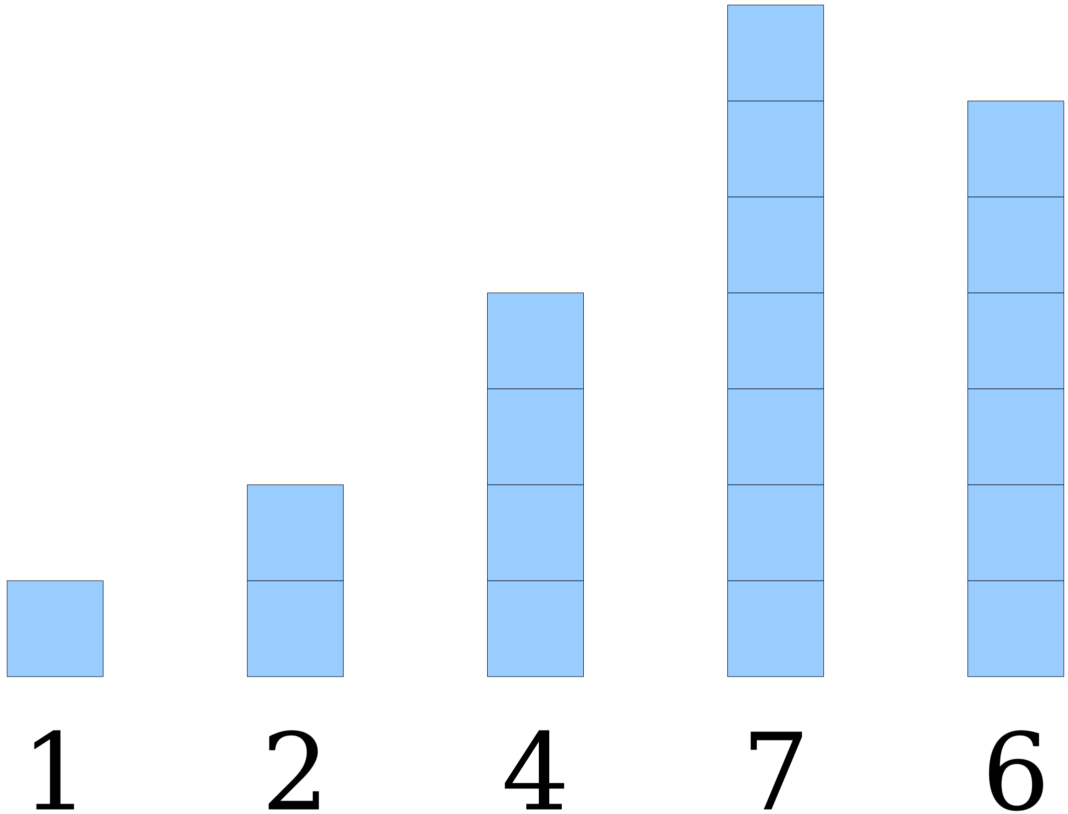
An Initial Idea: ***Selection Sort***



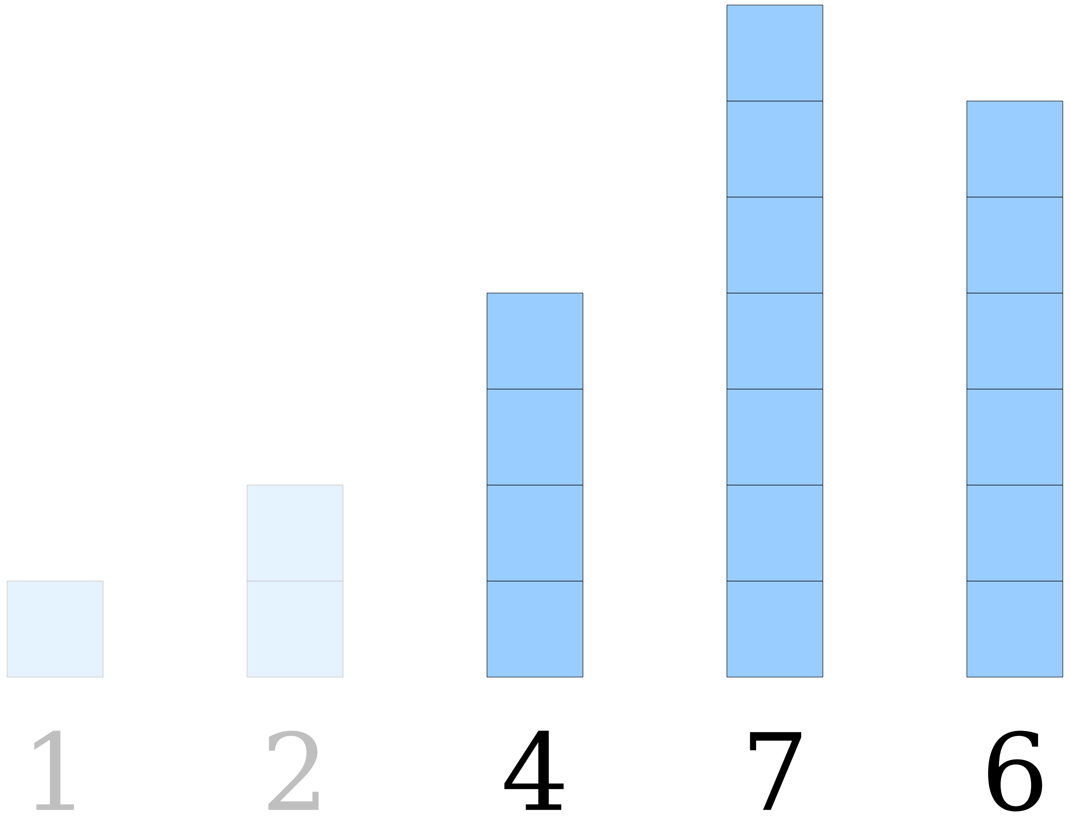
An Initial Idea: ***Selection Sort***



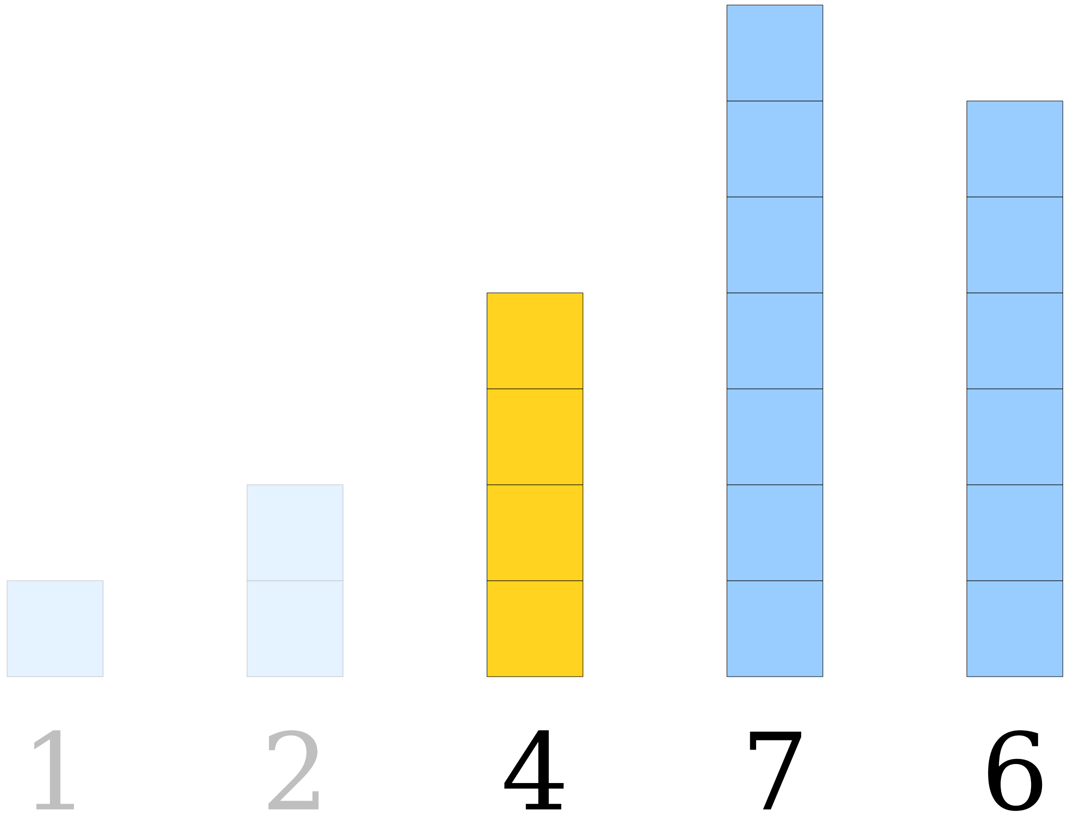
An Initial Idea: ***Selection Sort***



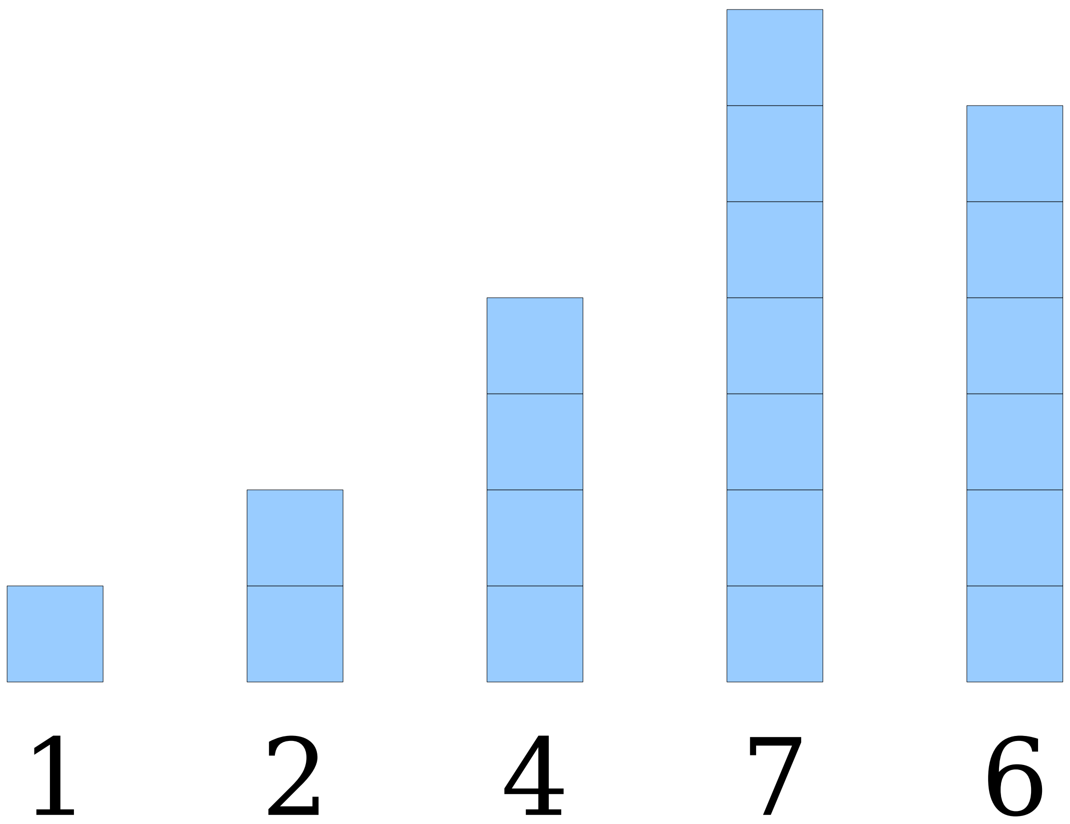
An Initial Idea: ***Selection Sort***



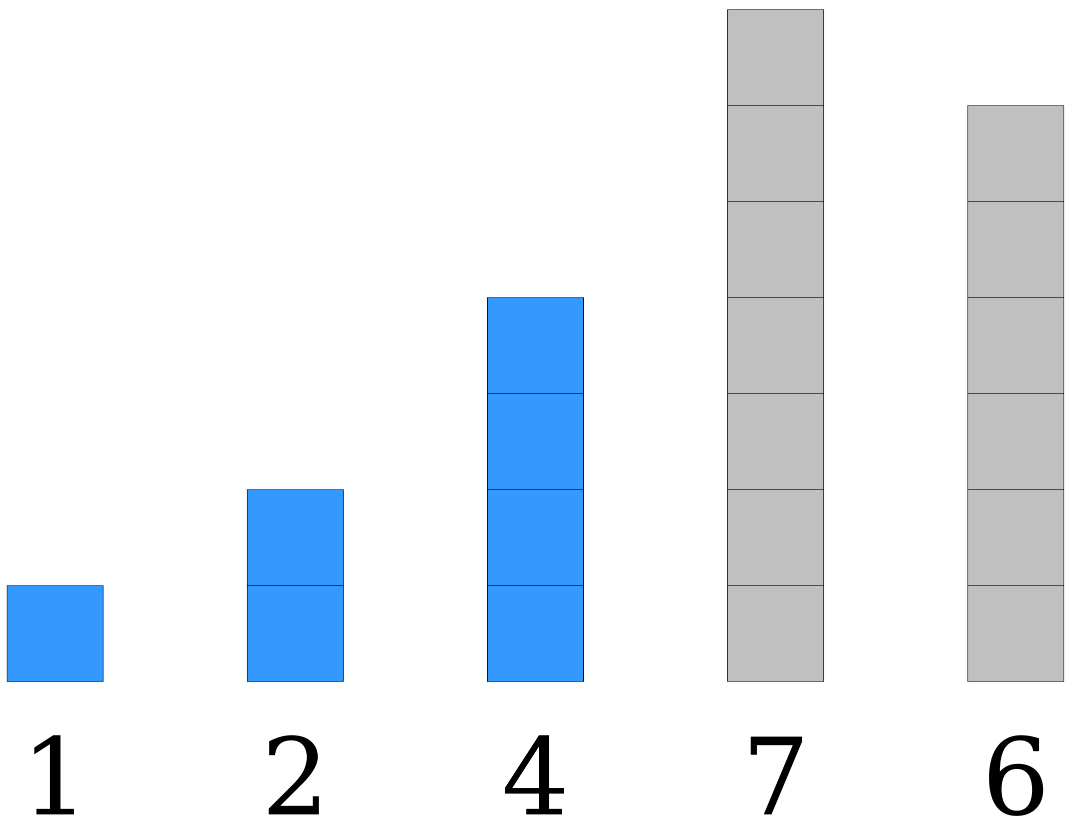
An Initial Idea: ***Selection Sort***



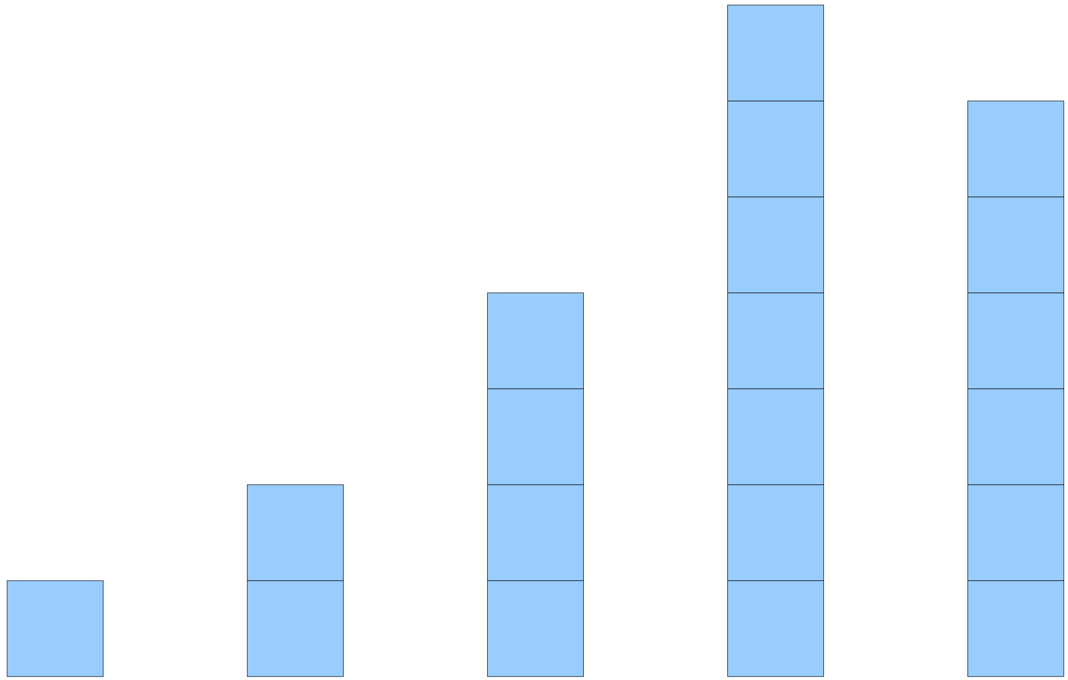
An Initial Idea: ***Selection Sort***



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

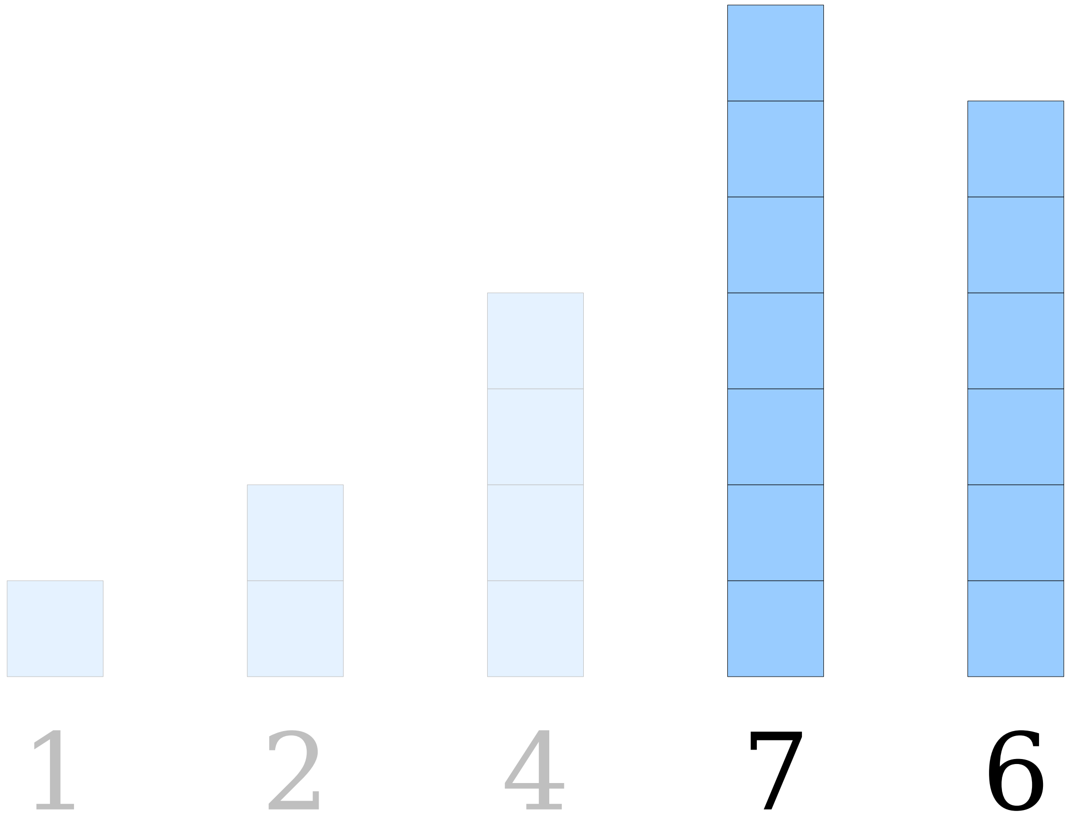


An Initial Idea: ***Selection Sort***

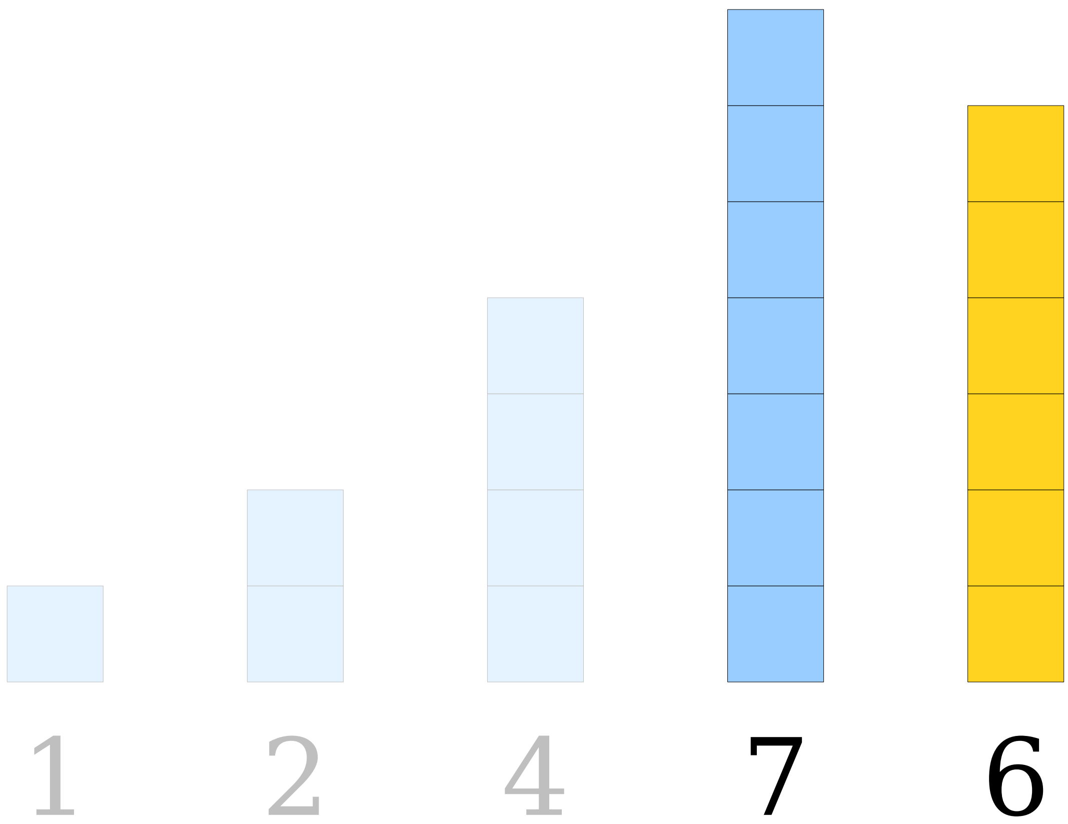


1 2 4 7 6

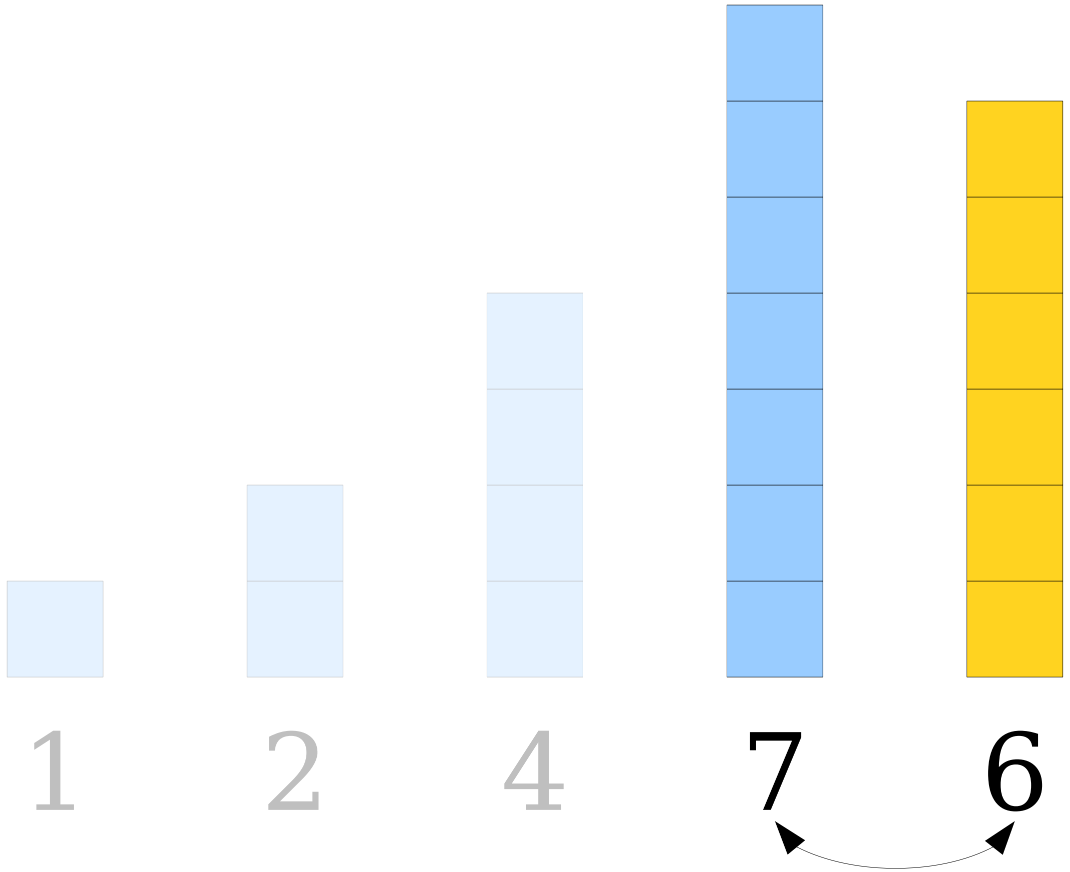
An Initial Idea: ***Selection Sort***



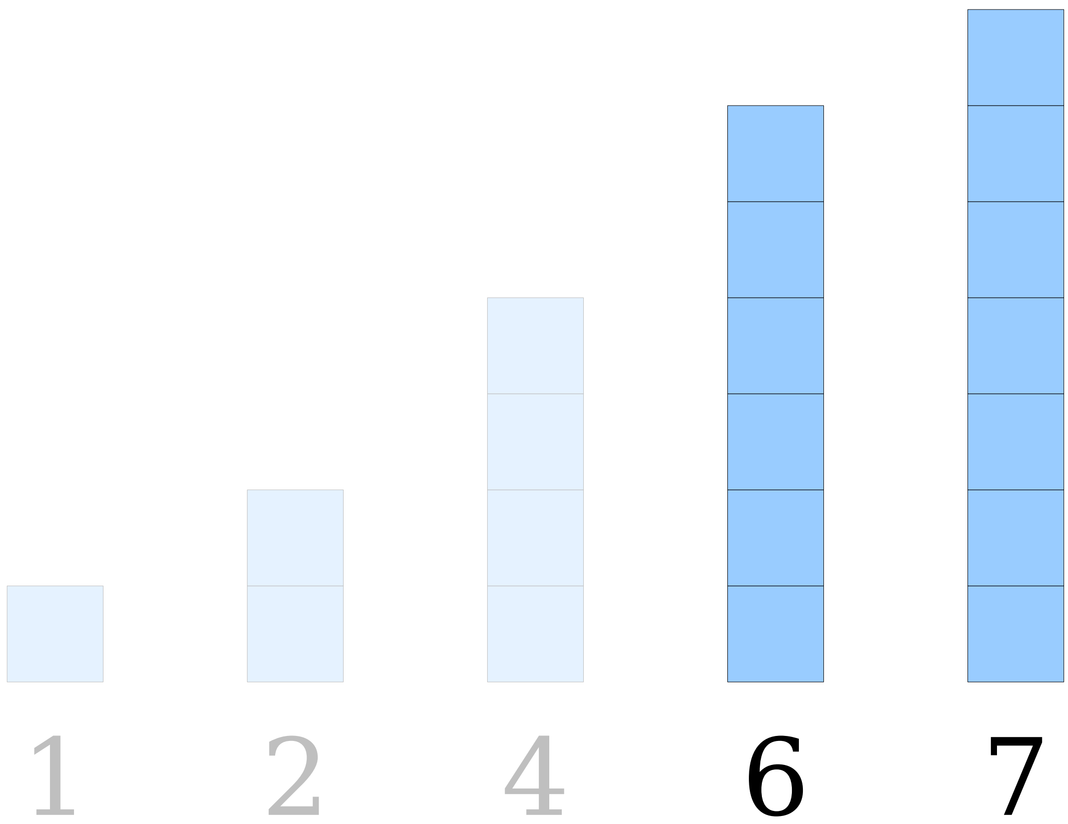
An Initial Idea: ***Selection Sort***



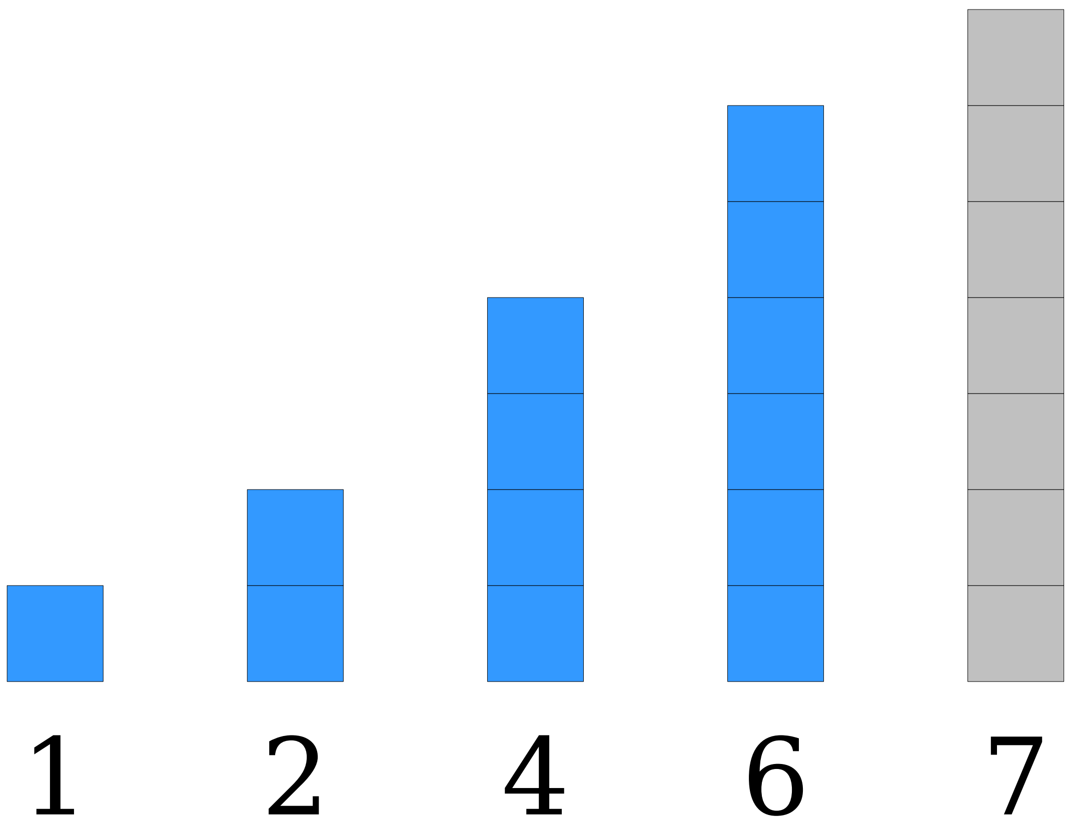
An Initial Idea: ***Selection Sort***



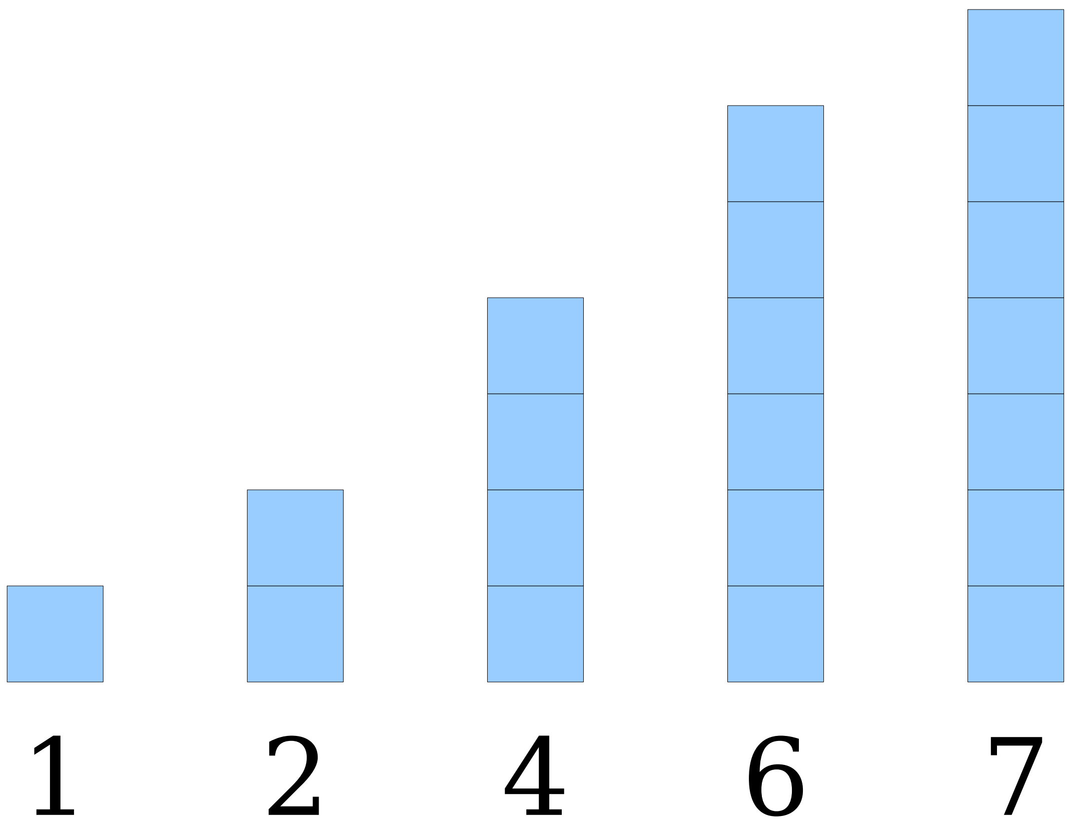
An Initial Idea: ***Selection Sort***



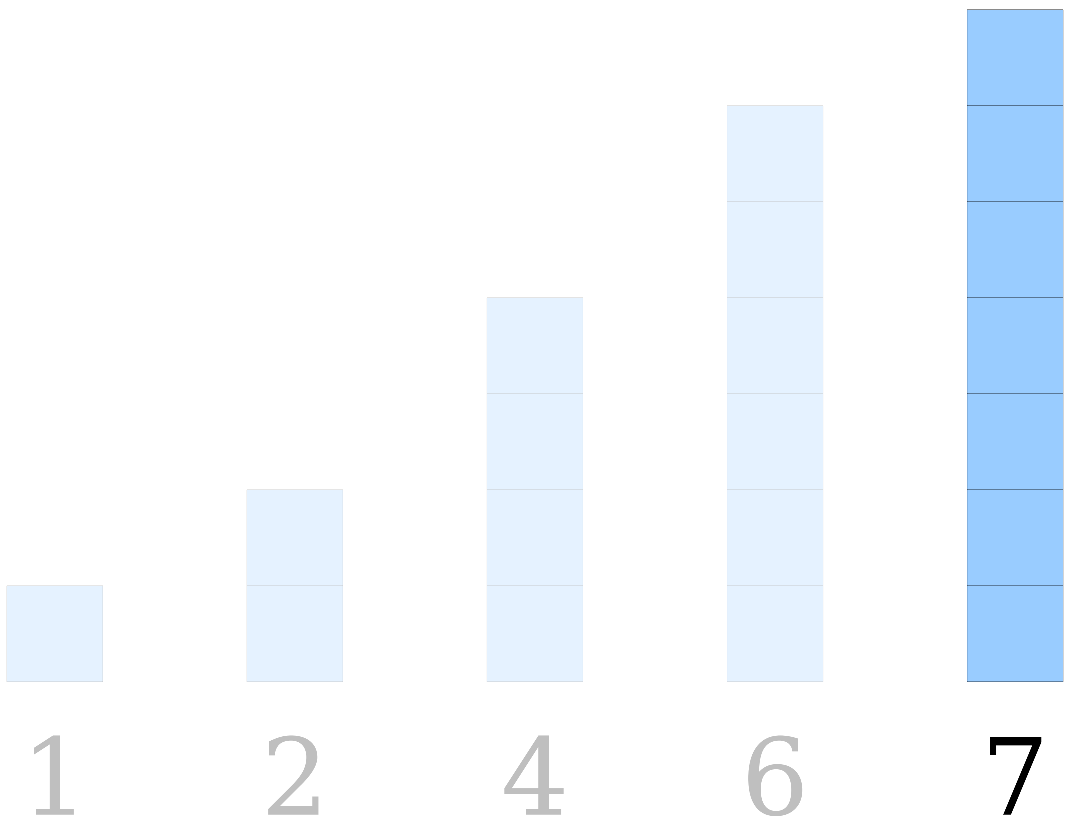
An Initial Idea: ***Selection Sort***



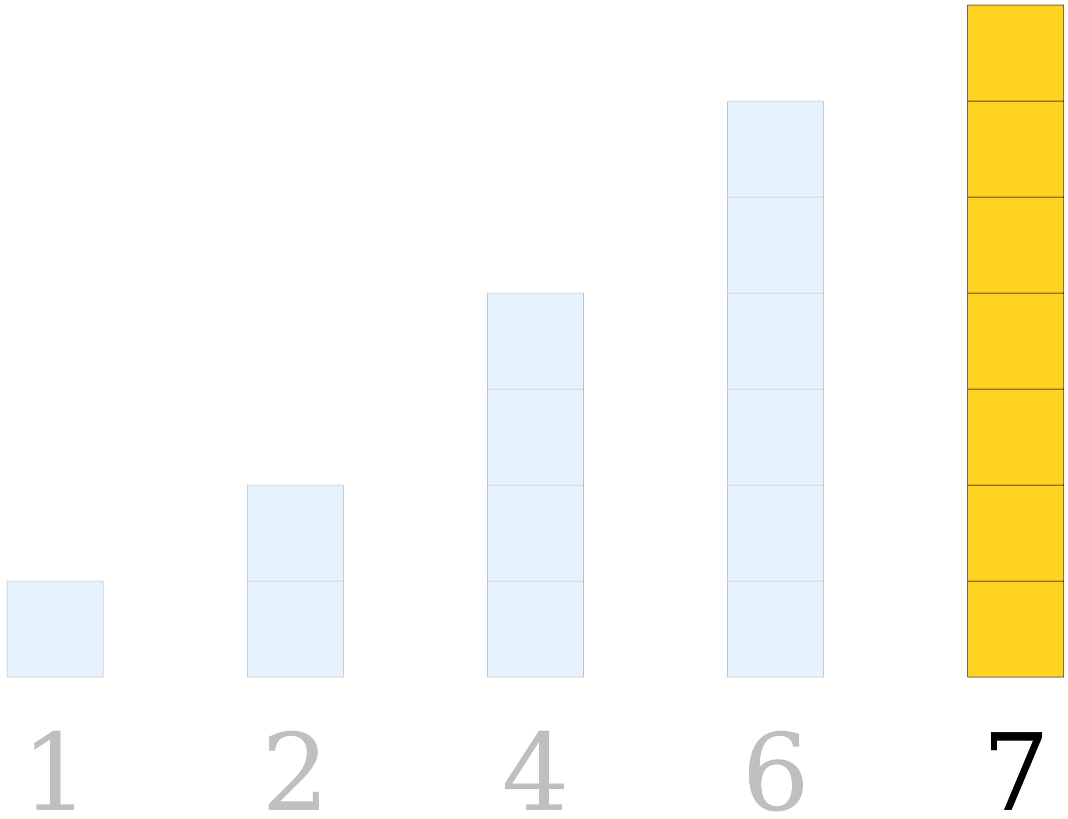
An Initial Idea: ***Selection Sort***



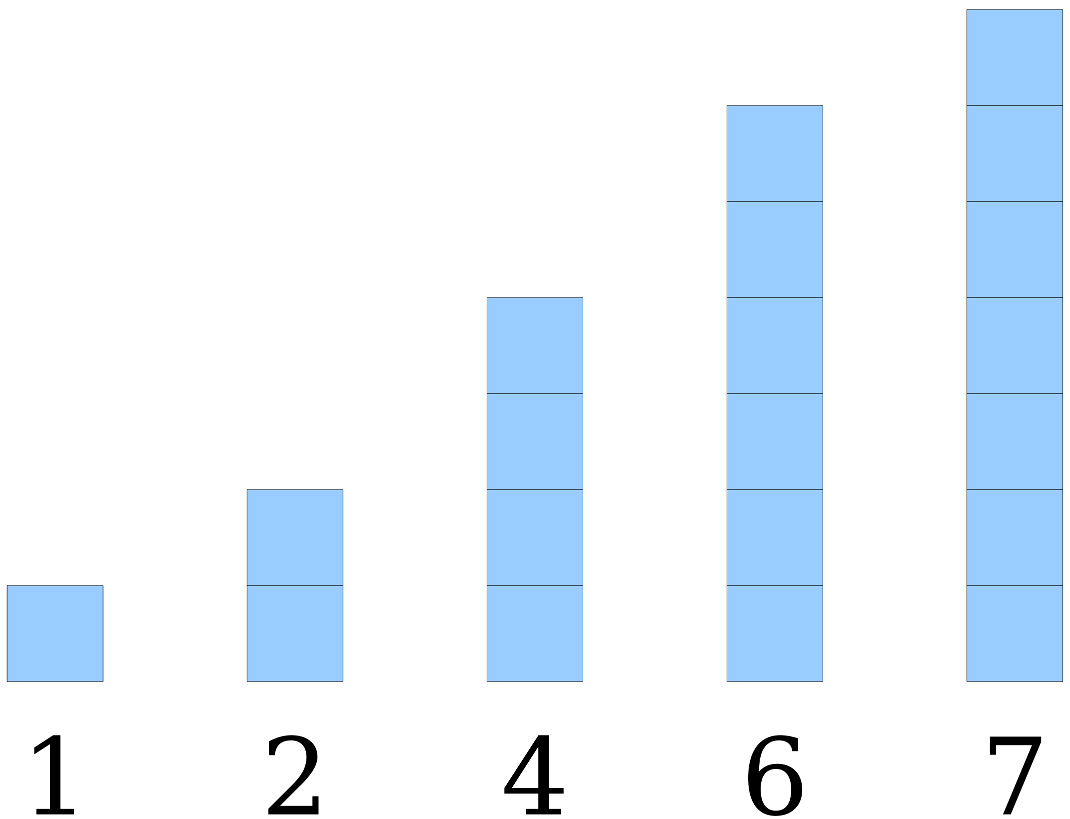
An Initial Idea: ***Selection Sort***



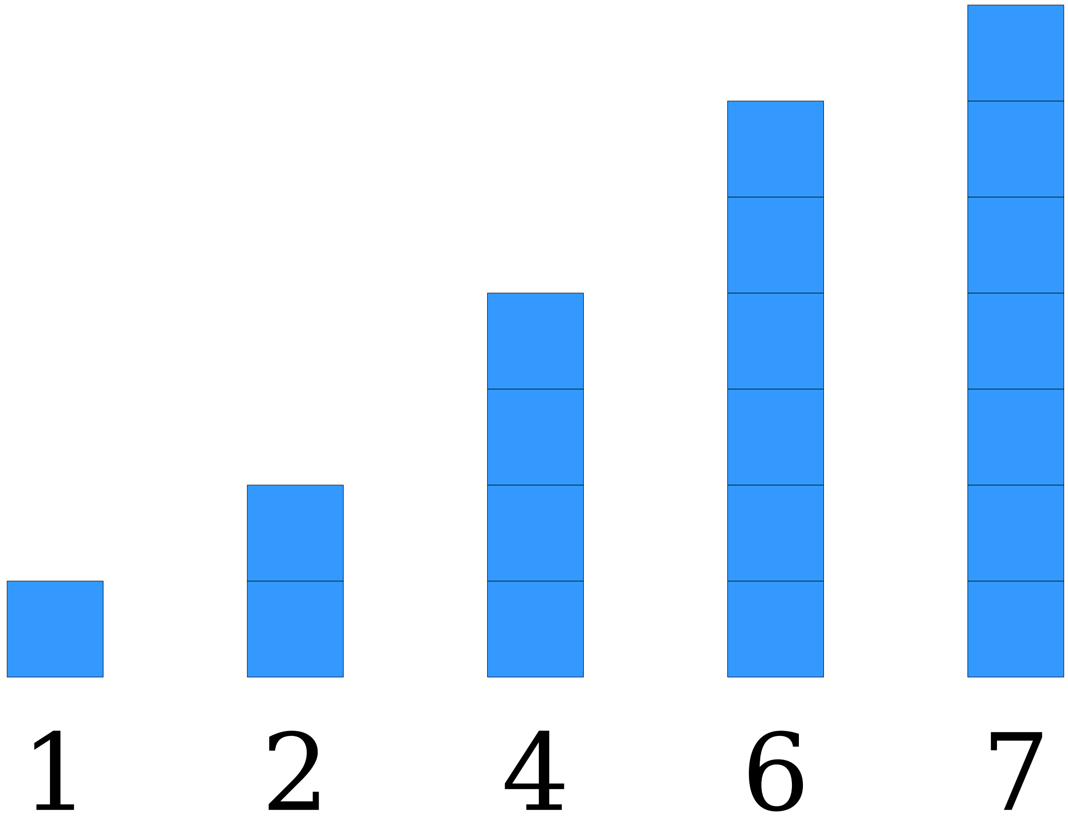
An Initial Idea: ***Selection Sort***



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

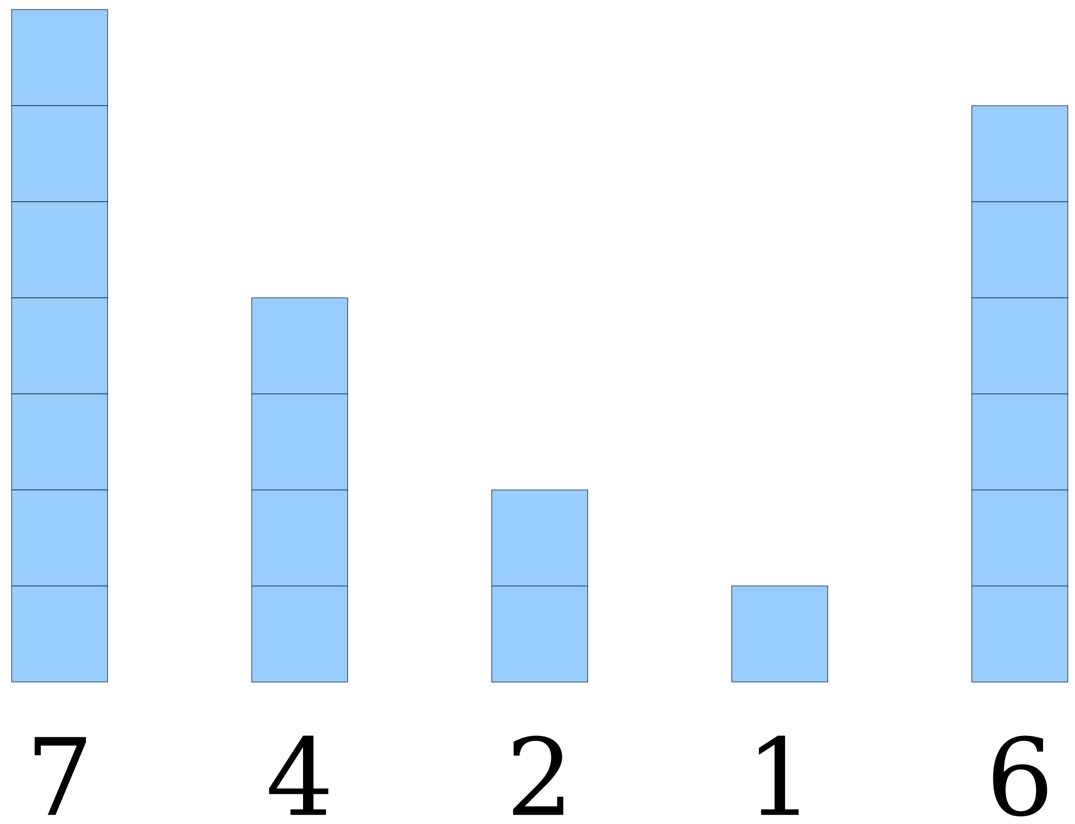


An Initial Idea: ***Selection Sort***

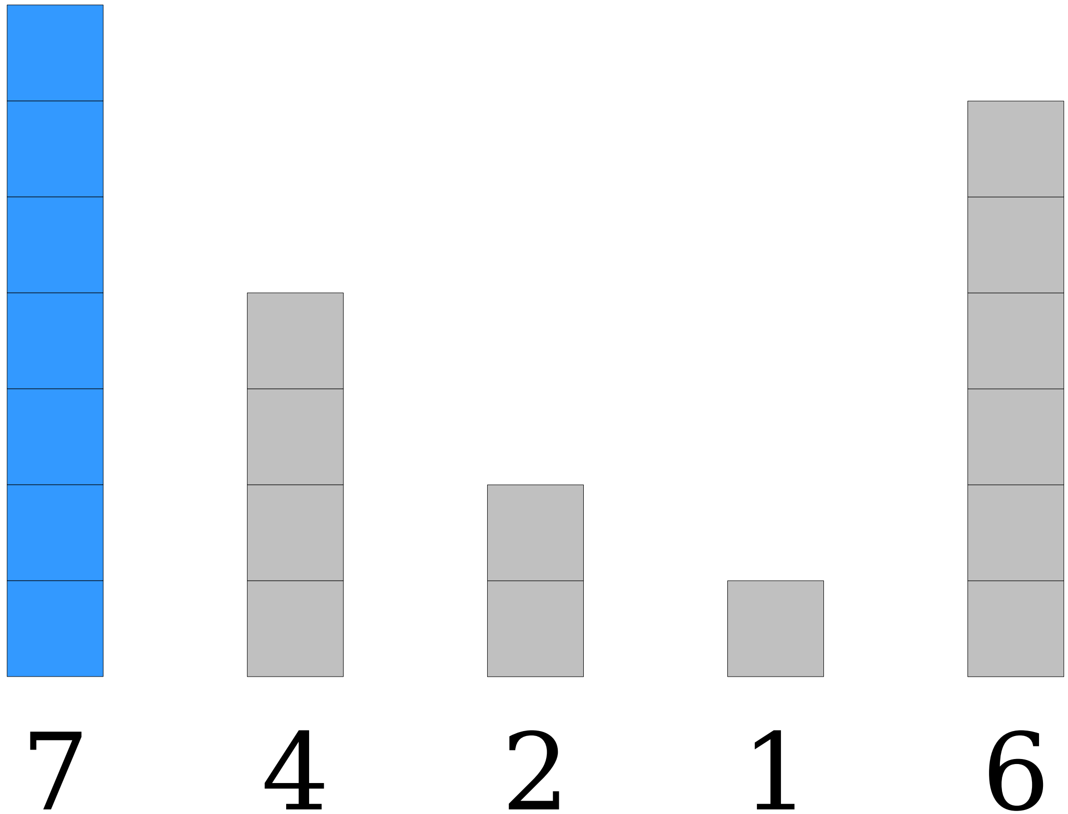


Our Next Idea: ***Insertion Sort***

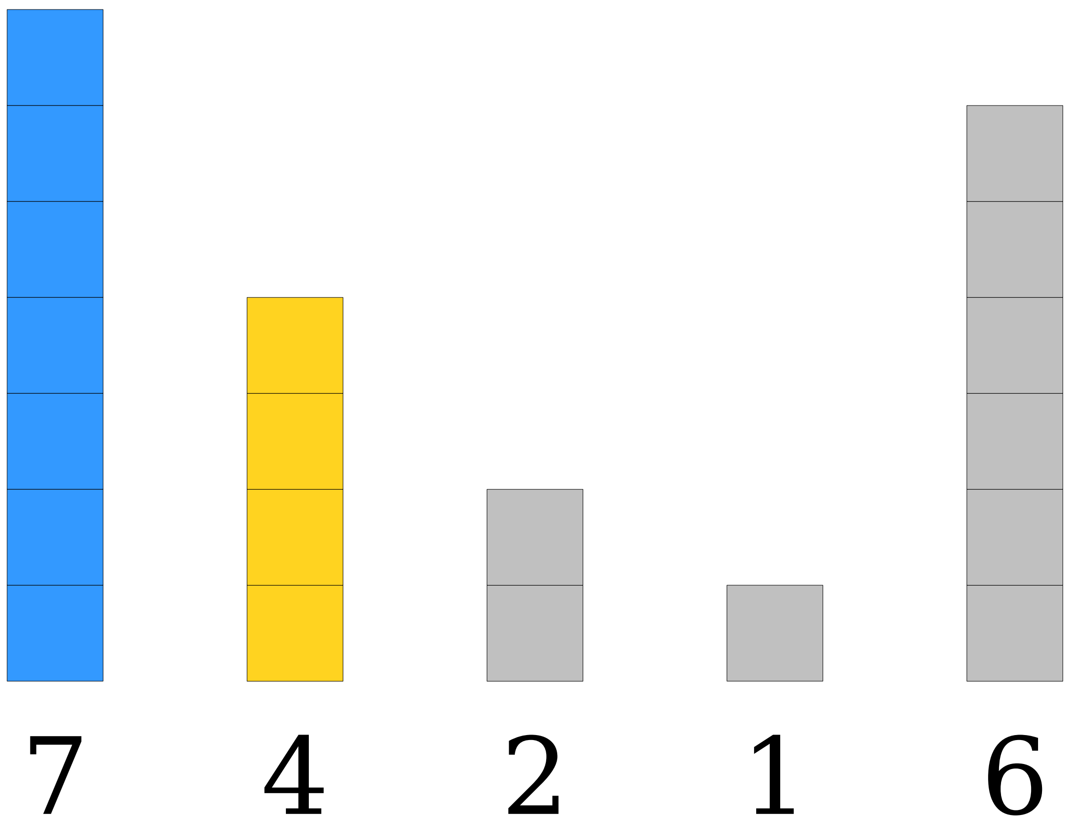
Our Next Idea: ***Insertion Sort***



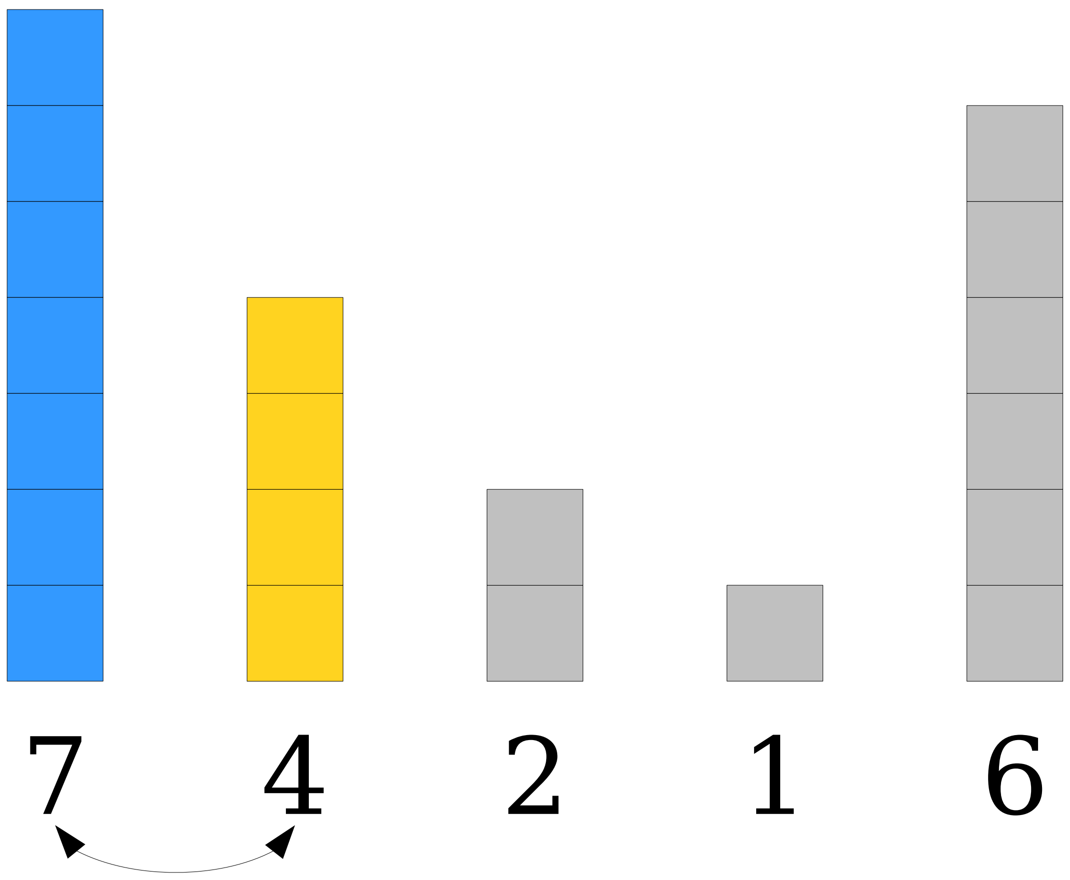
Our Next Idea: ***Insertion Sort***



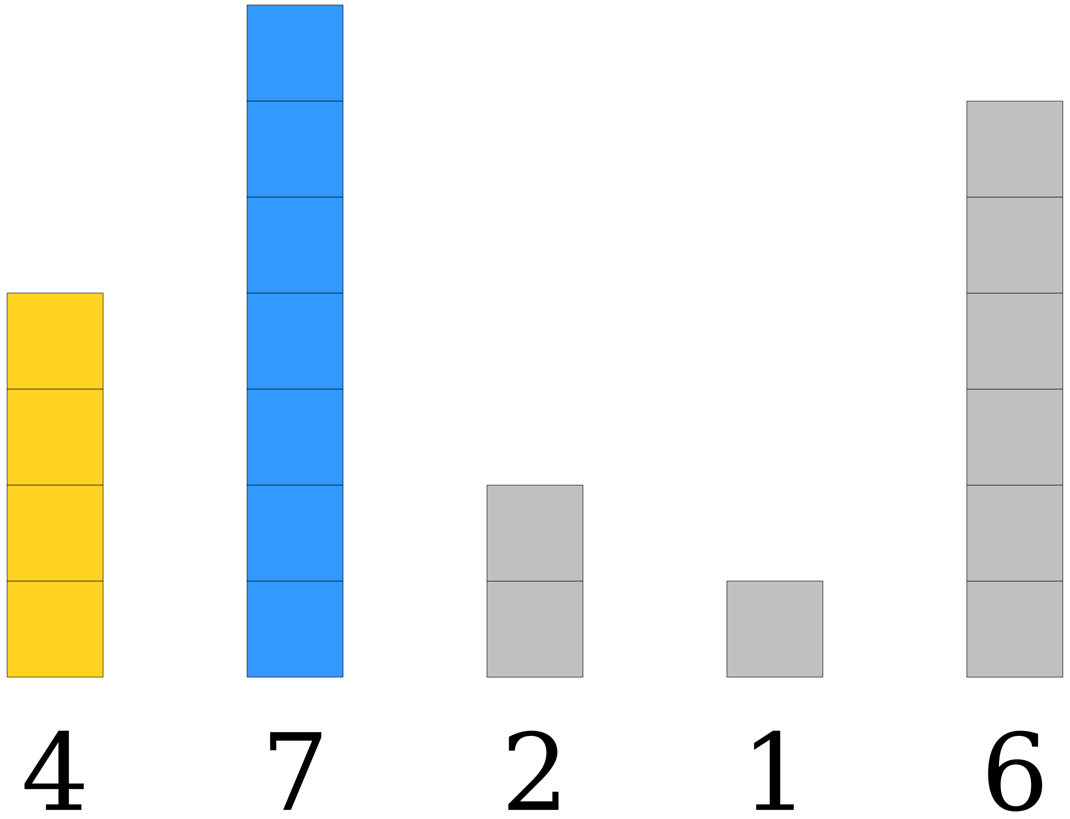
Our Next Idea: ***Insertion Sort***



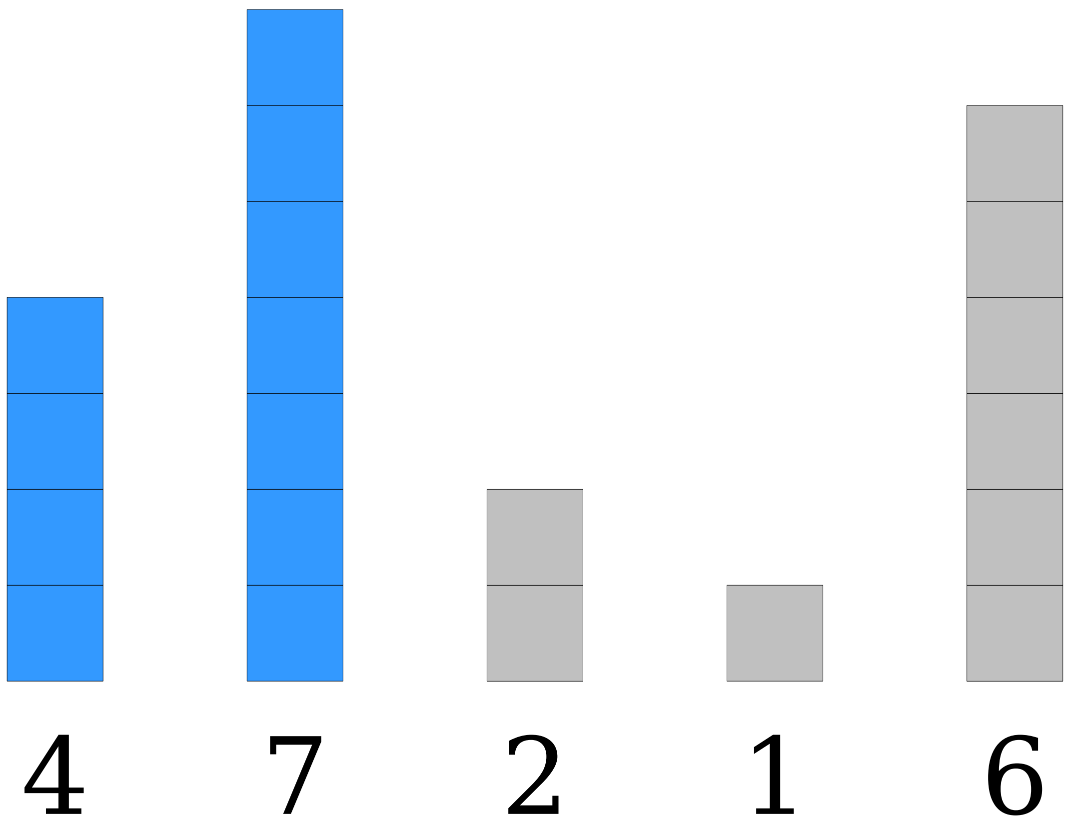
Our Next Idea: ***Insertion Sort***



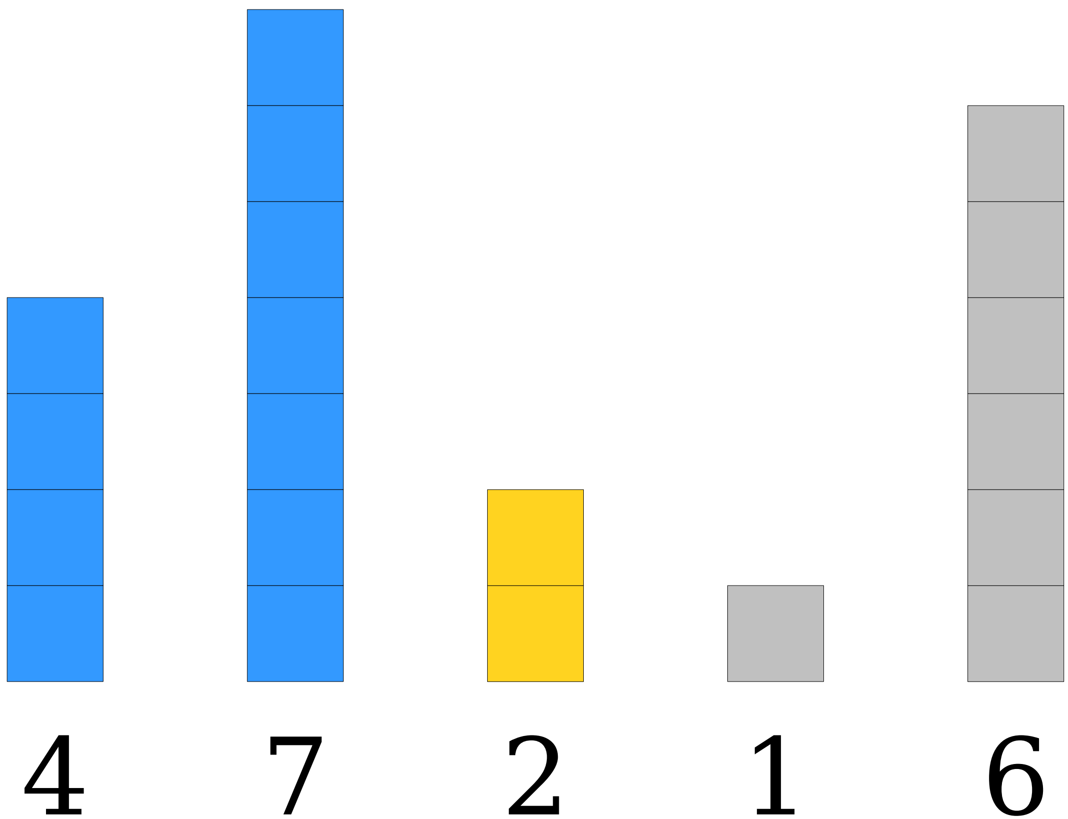
Our Next Idea: ***Insertion Sort***



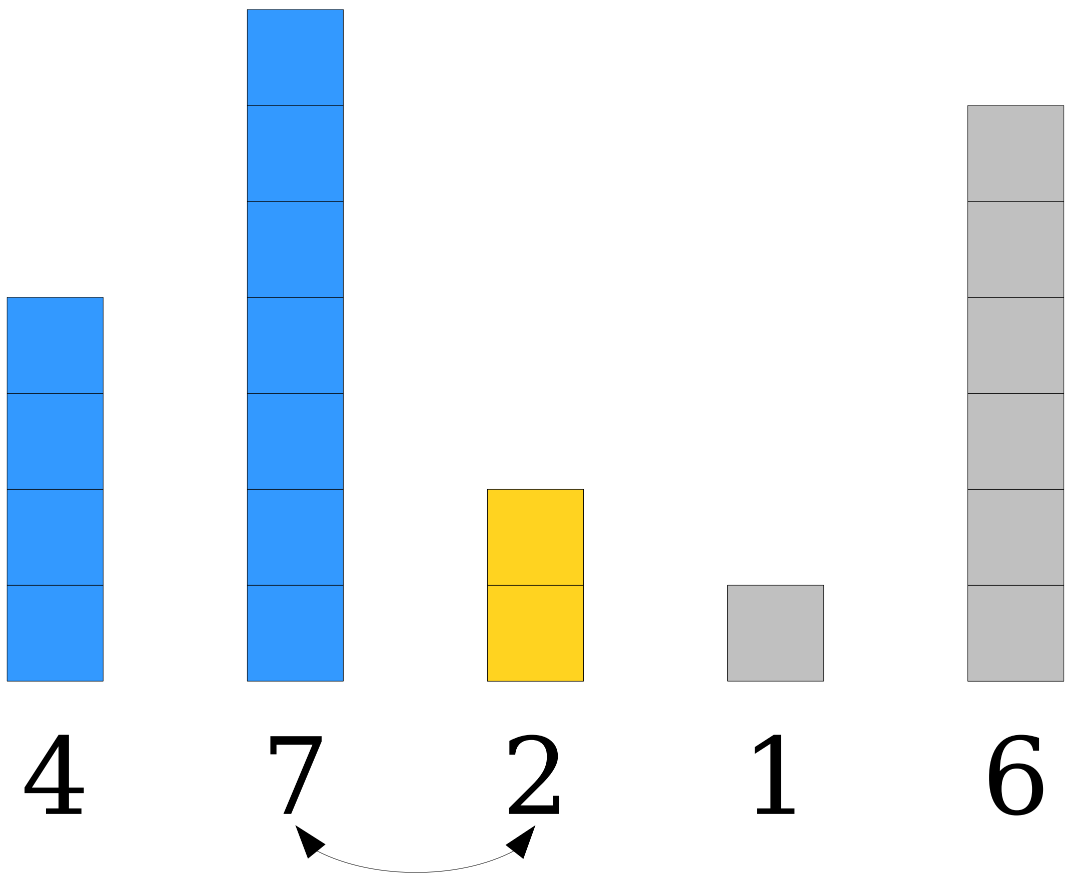
Our Next Idea: ***Insertion Sort***



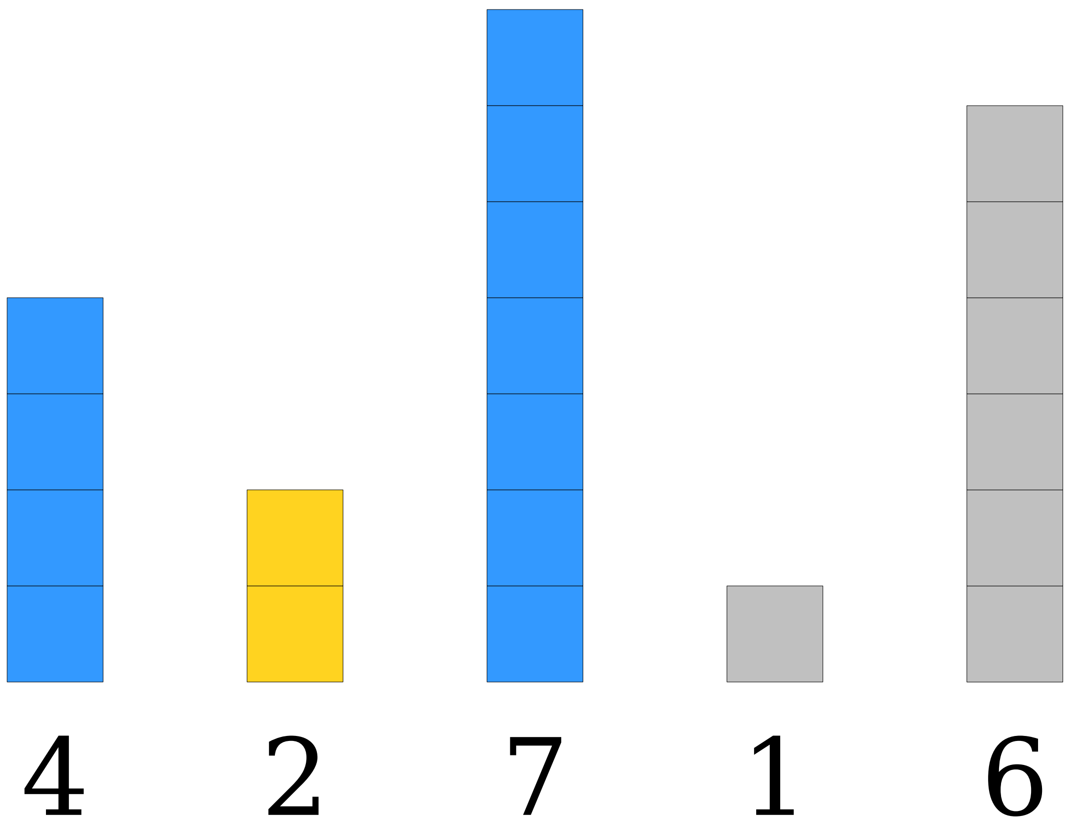
Our Next Idea: ***Insertion Sort***



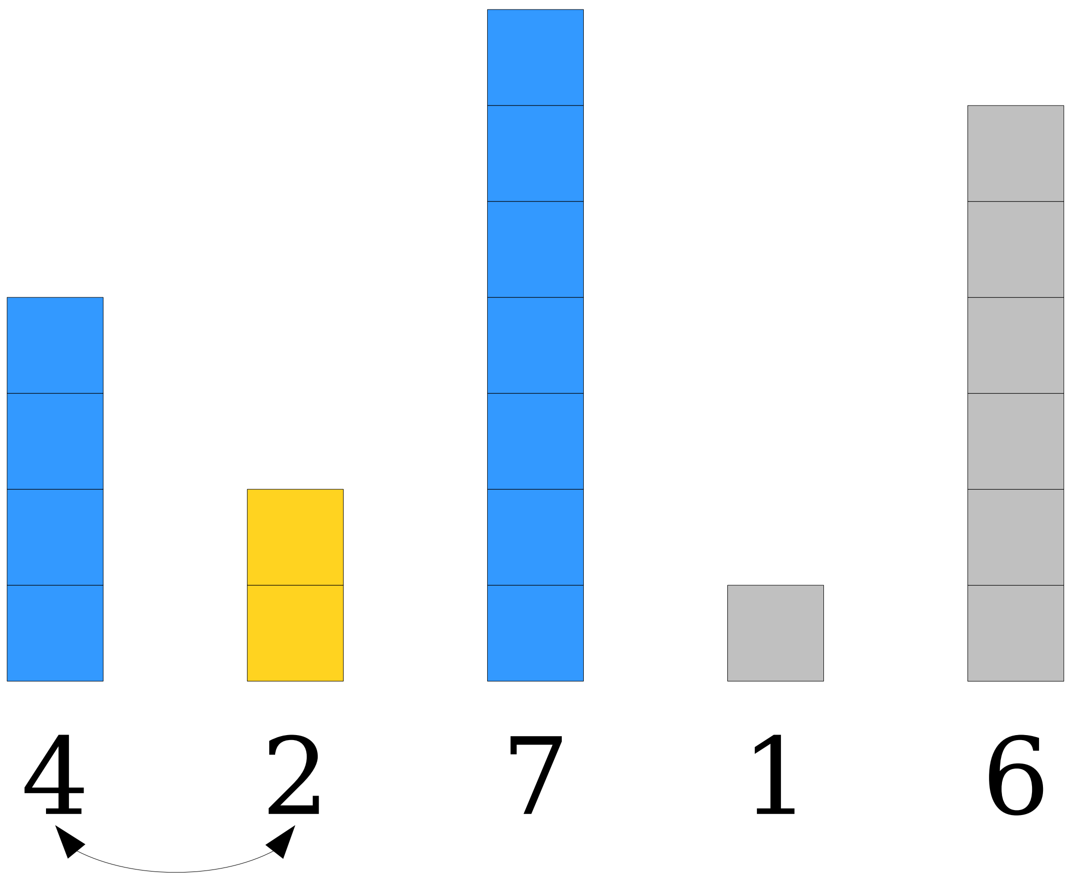
Our Next Idea: ***Insertion Sort***



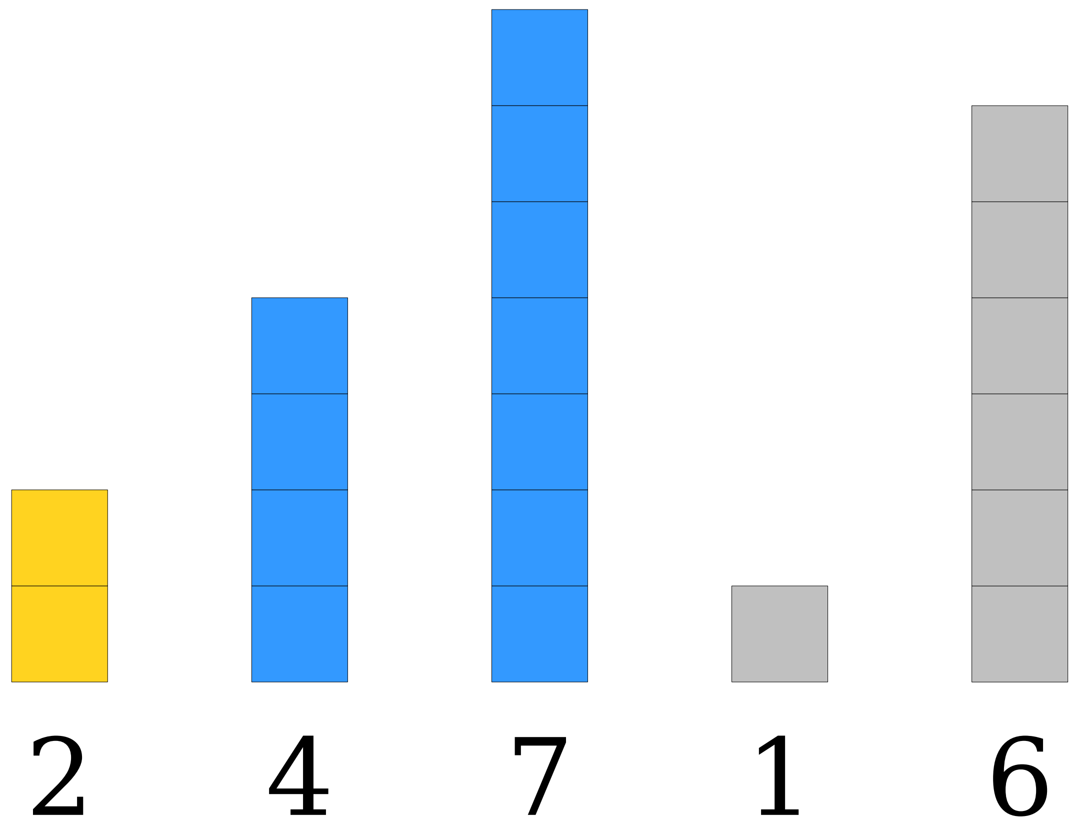
Our Next Idea: ***Insertion Sort***



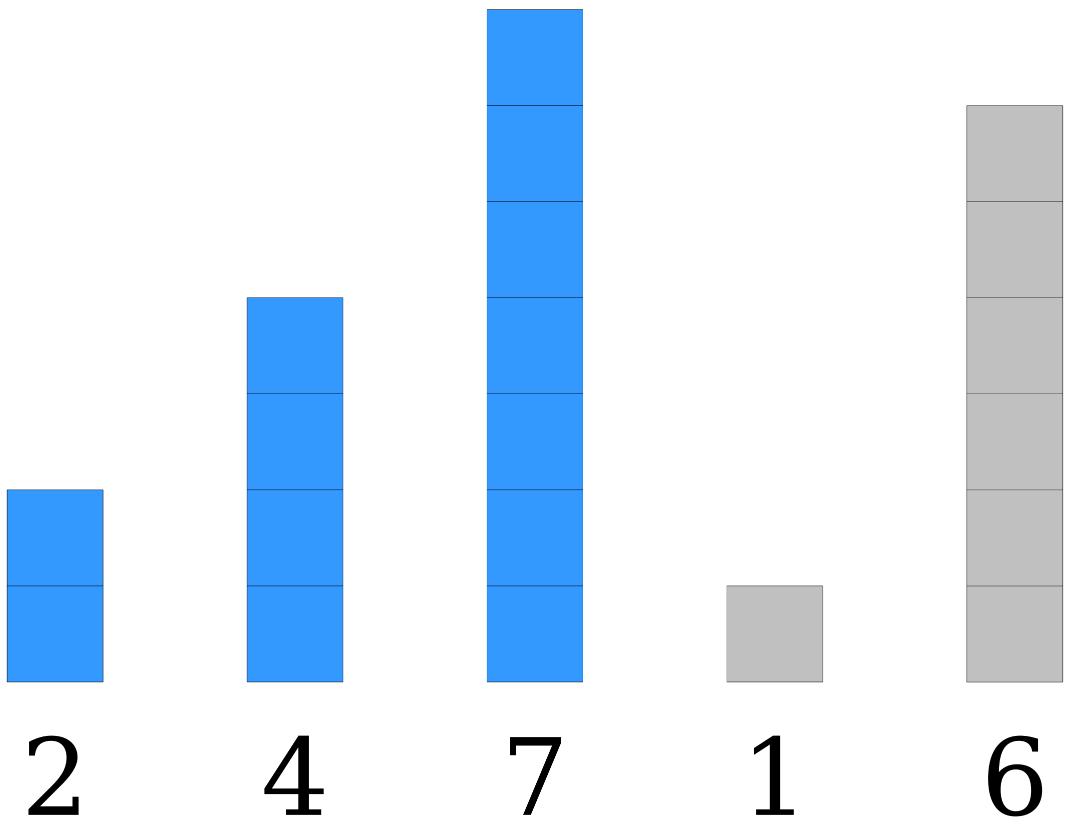
Our Next Idea: ***Insertion Sort***



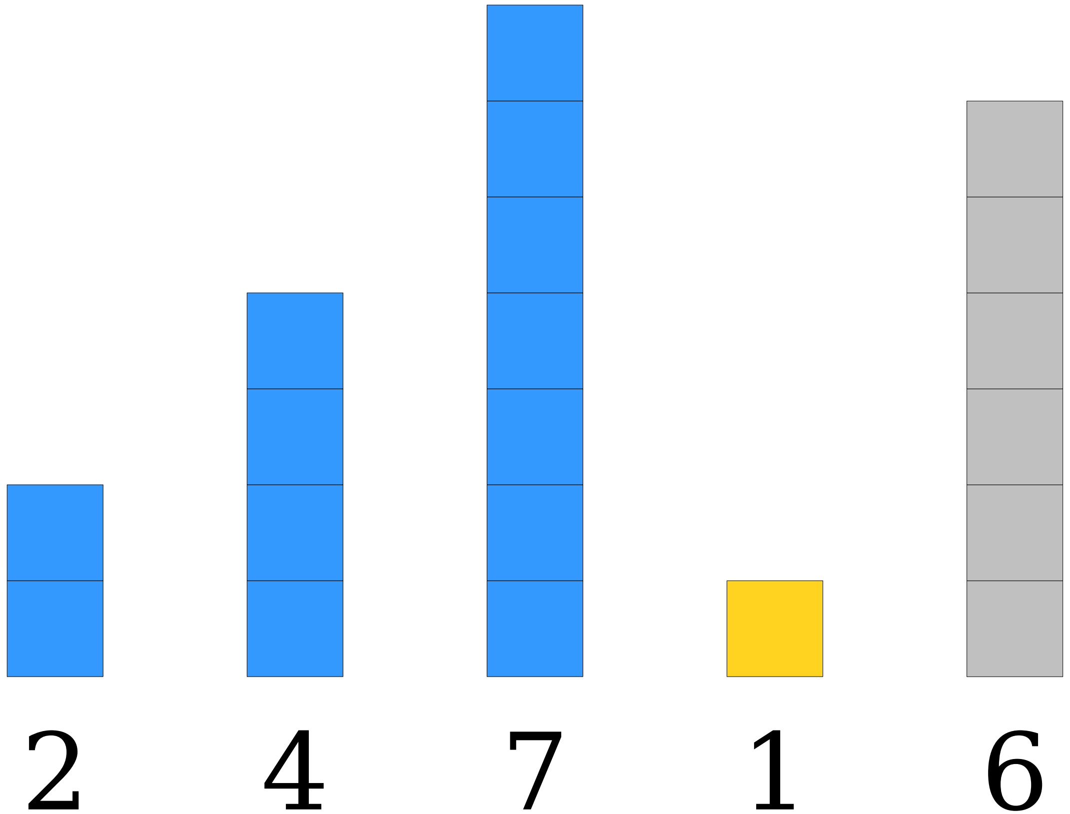
Our Next Idea: ***Insertion Sort***



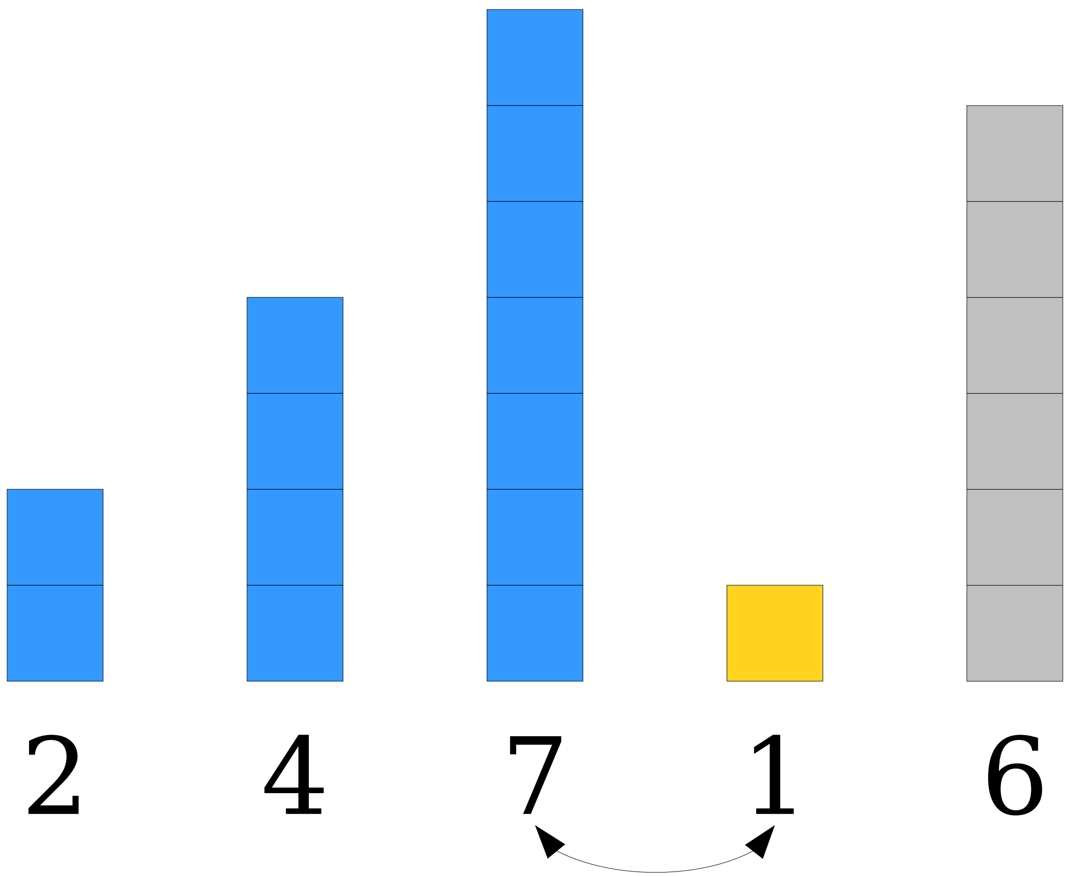
Our Next Idea: ***Insertion Sort***



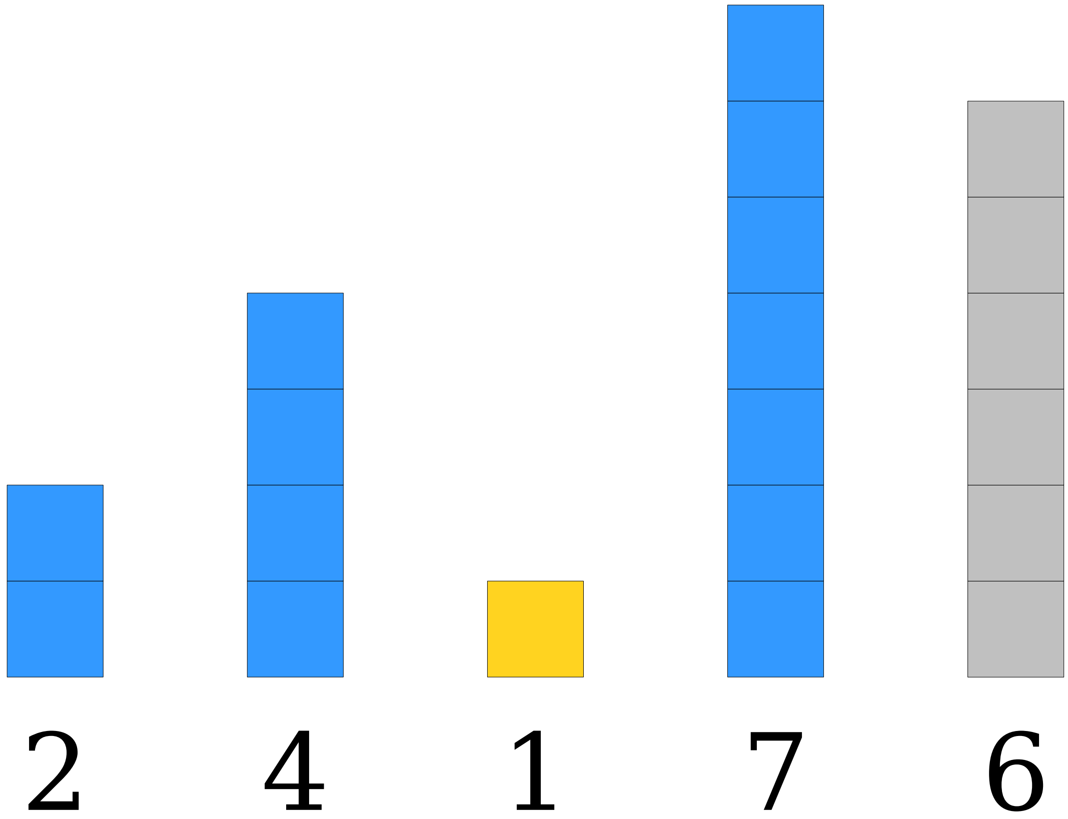
Our Next Idea: ***Insertion Sort***



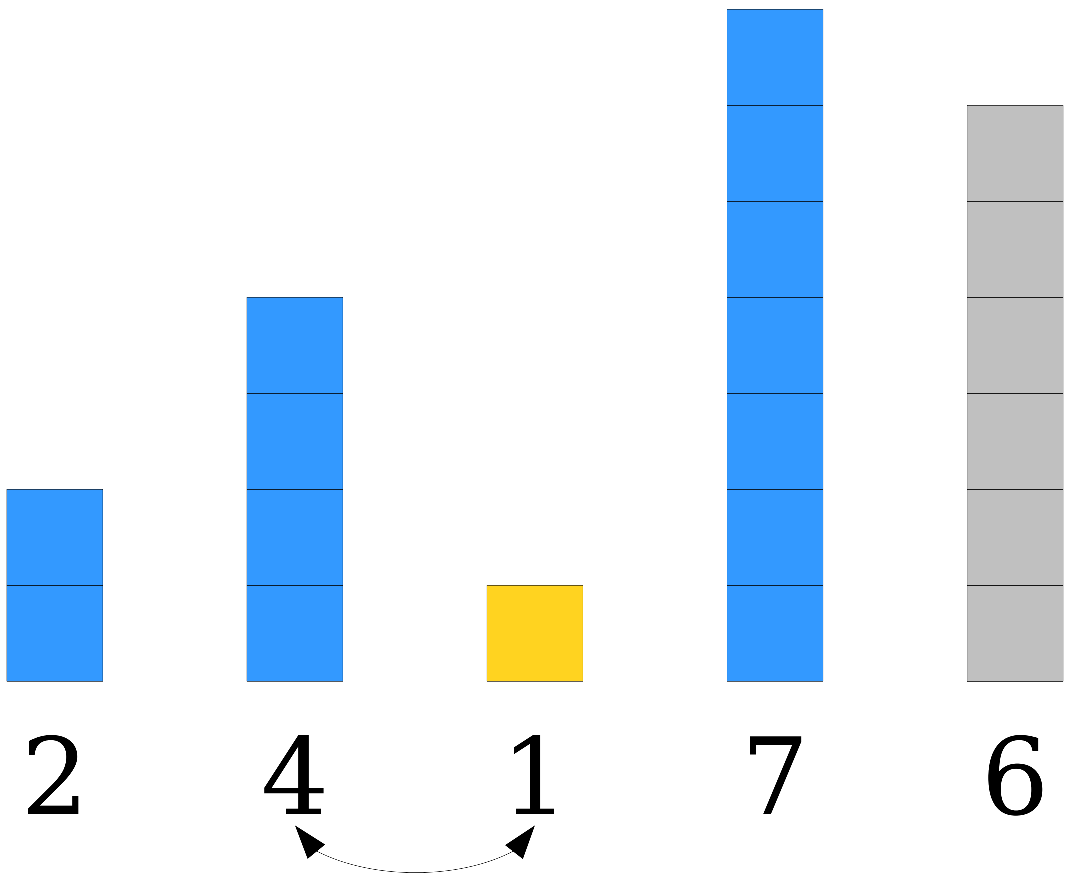
Our Next Idea: ***Insertion Sort***



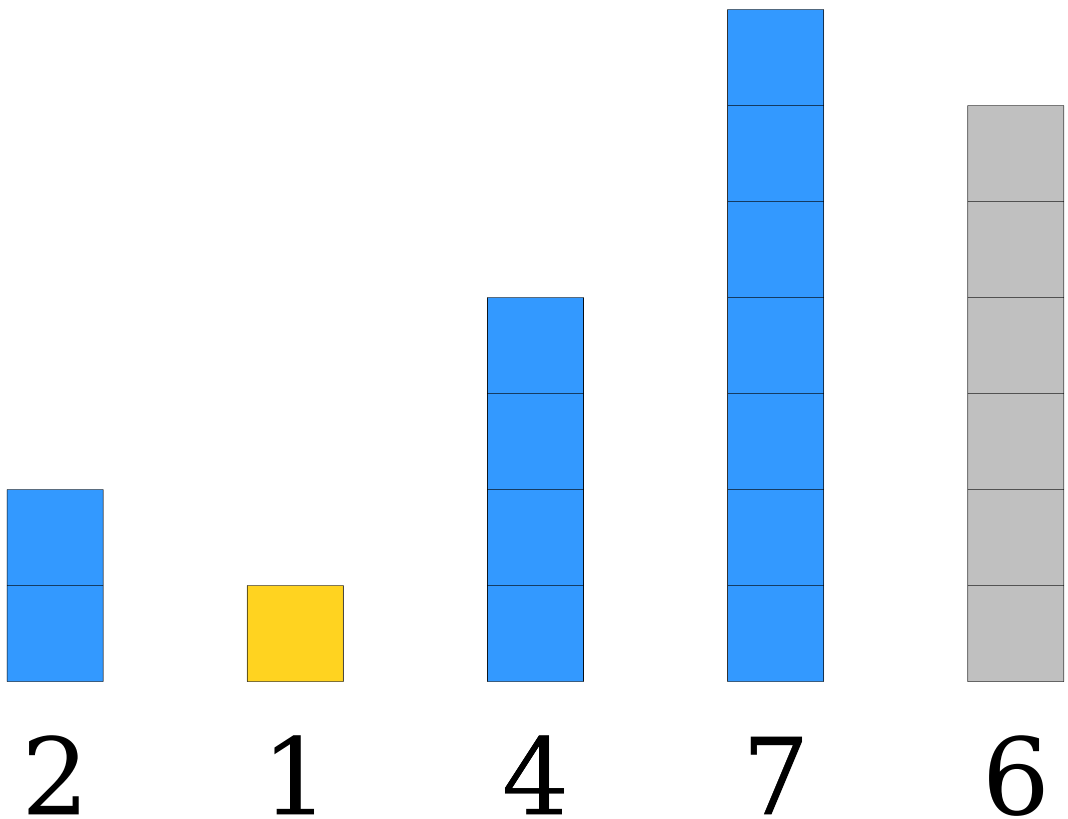
Our Next Idea: ***Insertion Sort***



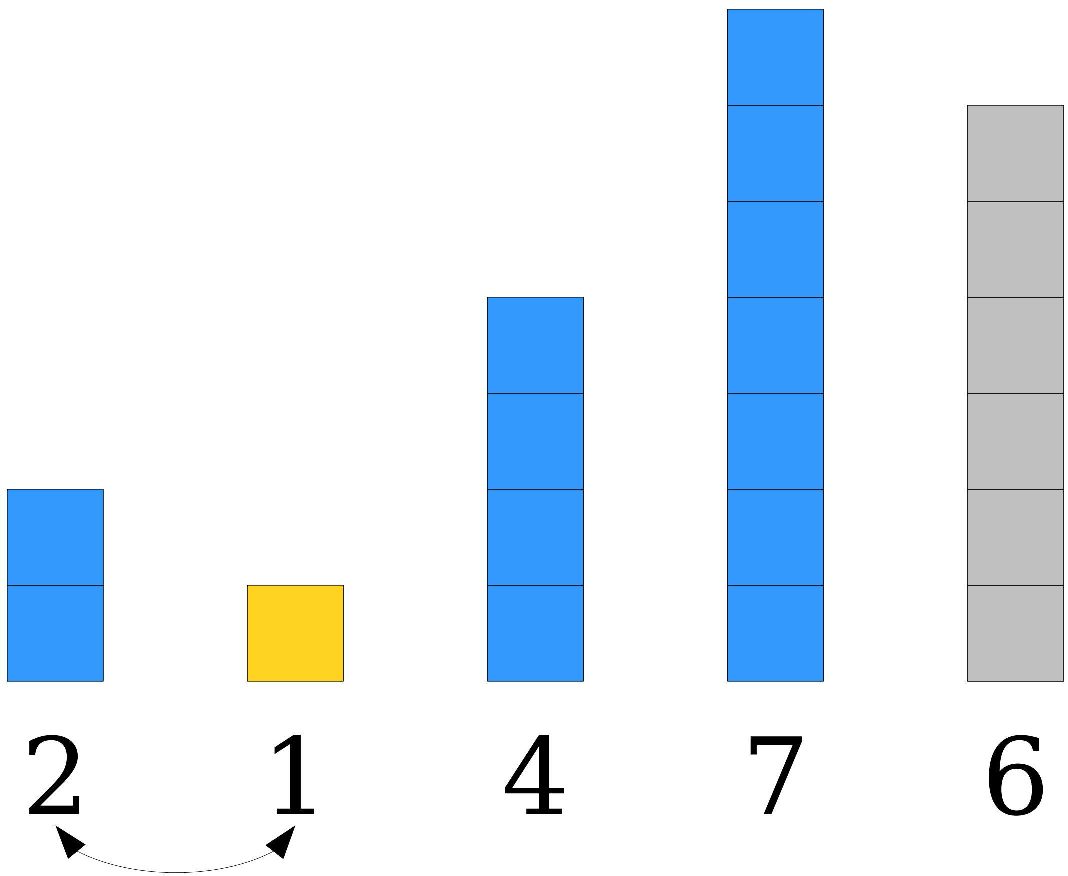
Our Next Idea: ***Insertion Sort***



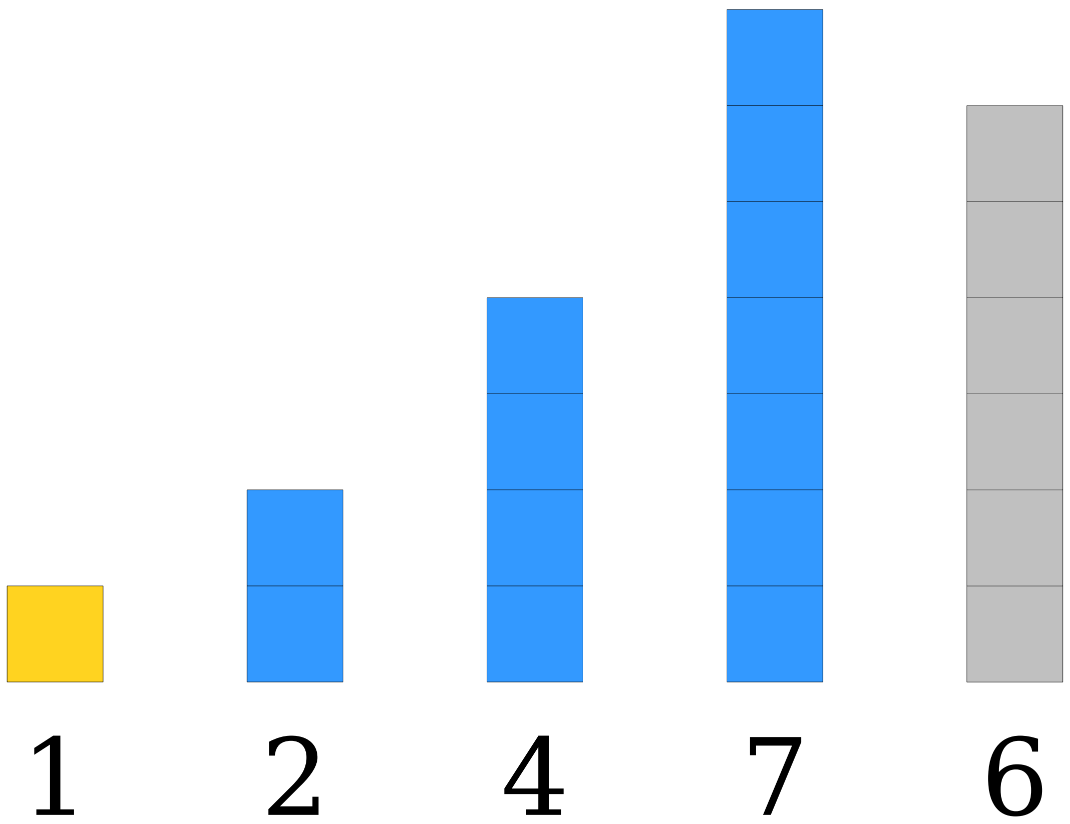
Our Next Idea: ***Insertion Sort***



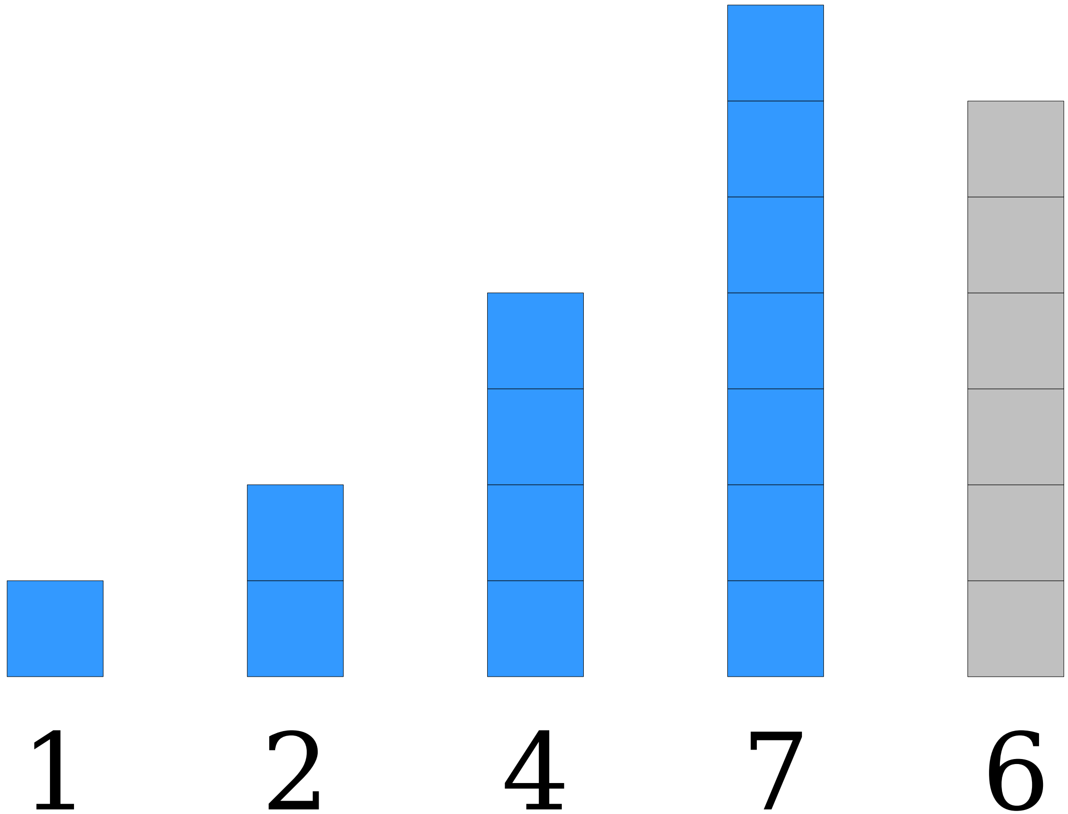
Our Next Idea: ***Insertion Sort***



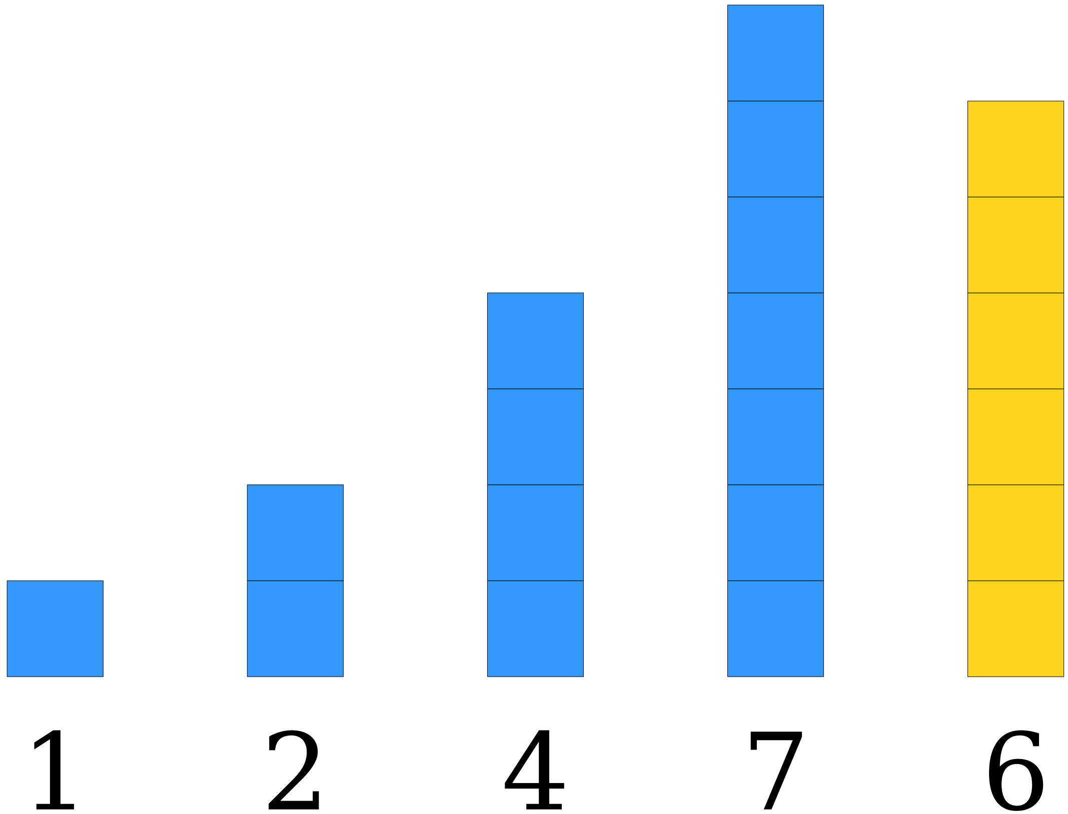
Our Next Idea: ***Insertion Sort***



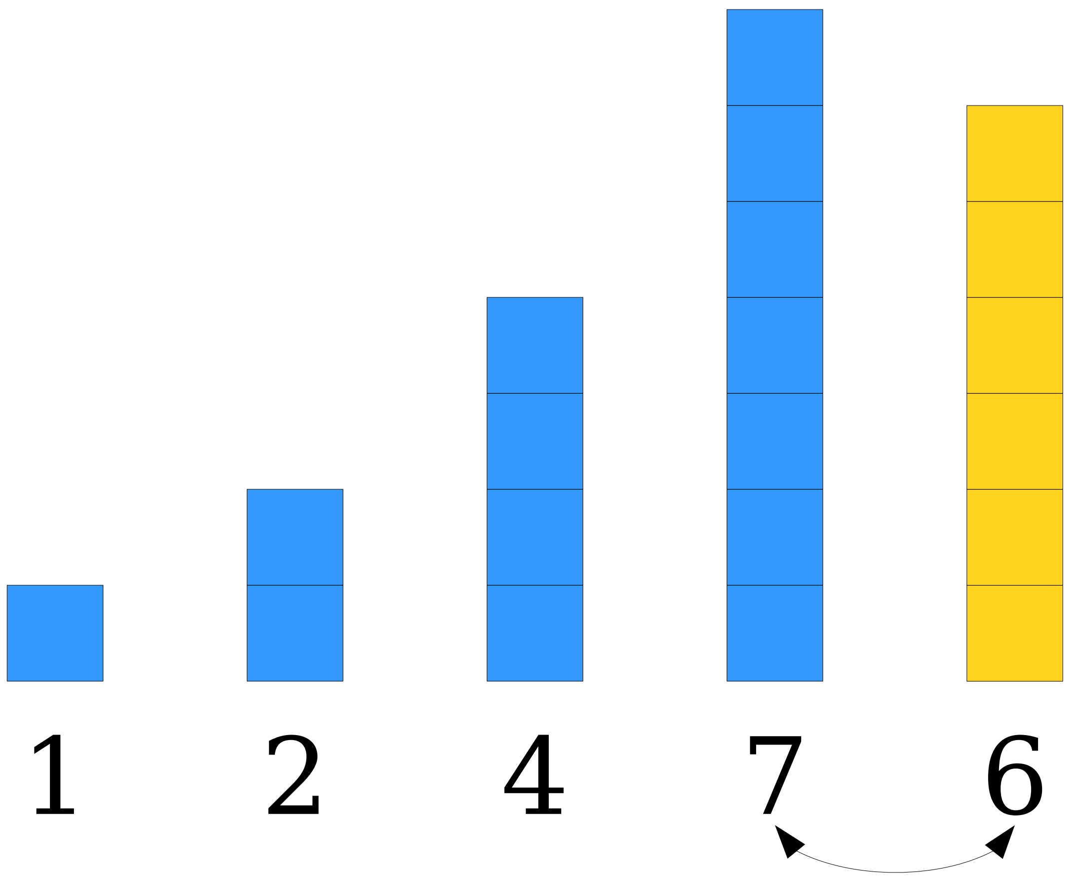
Our Next Idea: ***Insertion Sort***



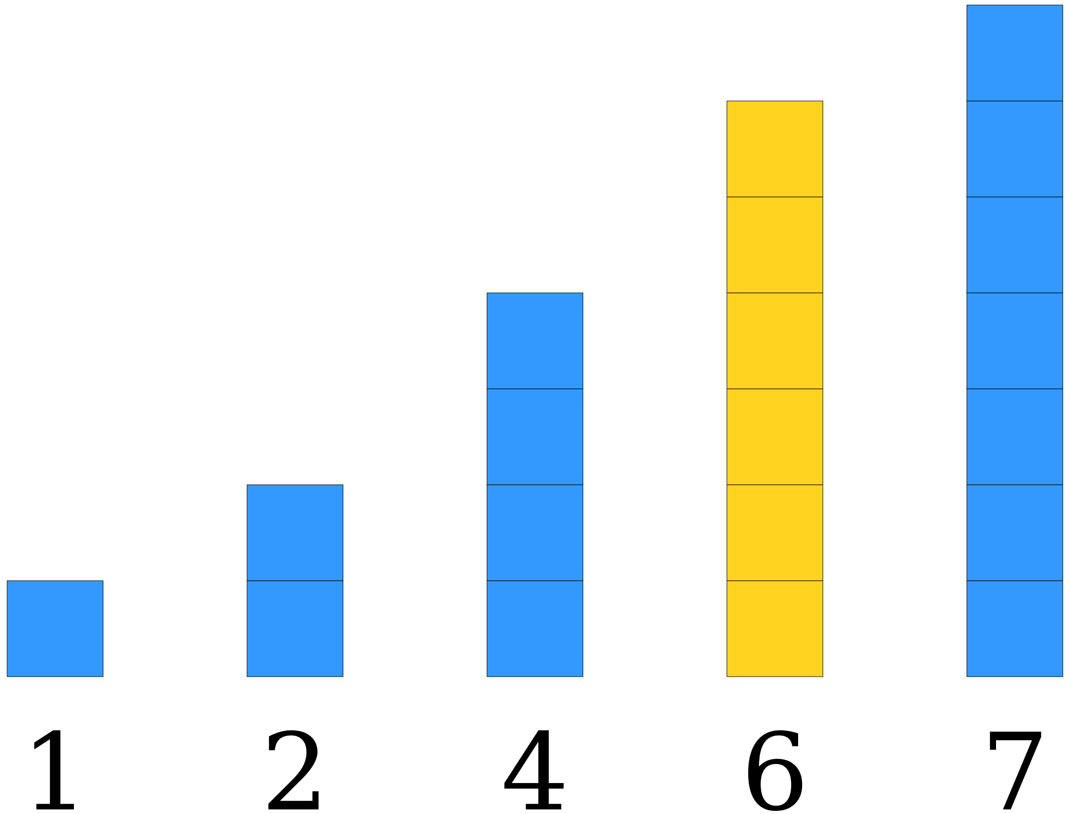
Our Next Idea: ***Insertion Sort***



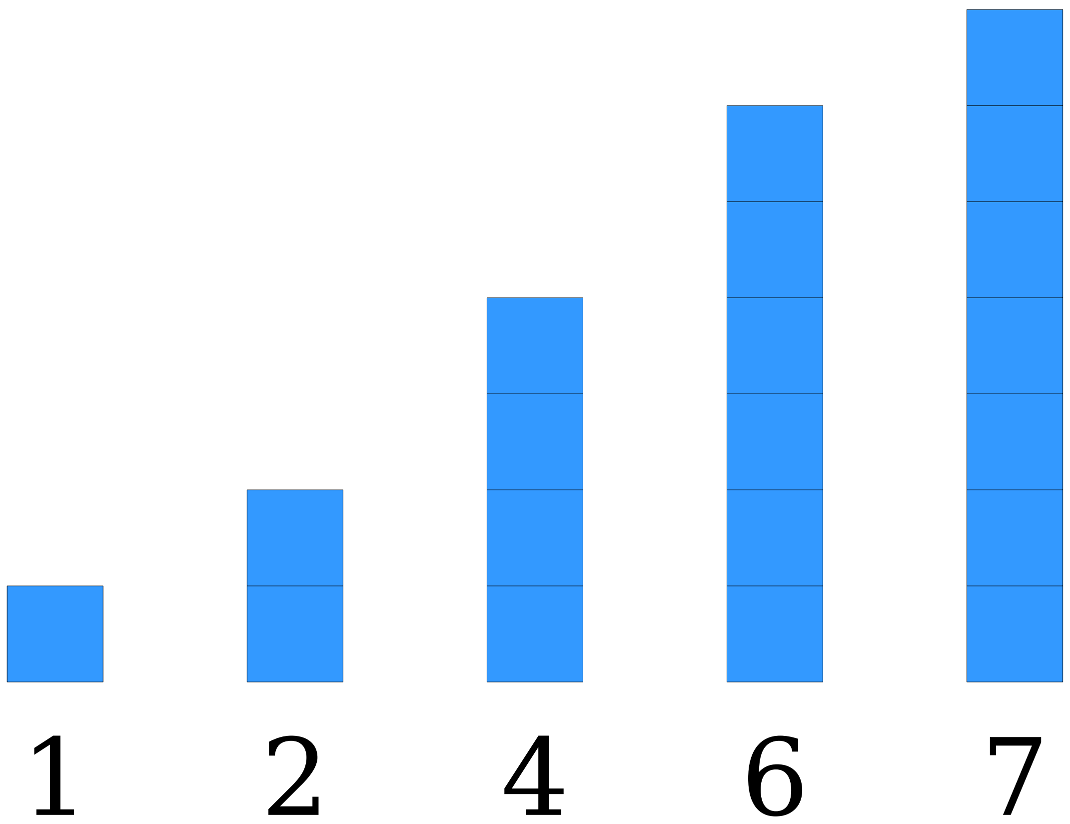
Our Next Idea: ***Insertion Sort***



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

5	10	8	11	1	13	12	4

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

Thinking About O(n^2)

$$\underbrace{\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline \mathtt{14} & \mathtt{6} & \mathtt{3} & \mathtt{9} & \mathtt{7} & \mathtt{16} & \mathtt{2} & \mathtt{15} & \mathtt{5} & \mathtt{10} & \mathtt{8} & \mathtt{11} & \mathtt{1} & \mathtt{13} & \mathtt{12} & \mathtt{4} \\ \hline \end{array}}_{\mathrm{T}(n)}$$

$$\underbrace{\begin{array}{|c|c|c|c|c|c|c|c|}\hline \mathtt{14} & \mathtt{6} & \mathtt{3} & \mathtt{9} & \mathtt{7} & \mathtt{16} & \mathtt{2} & \mathtt{15} \\ \hline \end{array}}_{\frac{1}{4}\mathrm{T}(n)} \quad \underbrace{\begin{array}{|c|c|c|c|c|c|c|c|}\hline \mathtt{5} & \mathtt{10} & \mathtt{8} & \mathtt{11} & \mathtt{1} & \mathtt{13} & \mathtt{12} & \mathtt{4} \\ \hline \end{array}}_{\frac{1}{4}\mathrm{T}(n)}$$

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

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

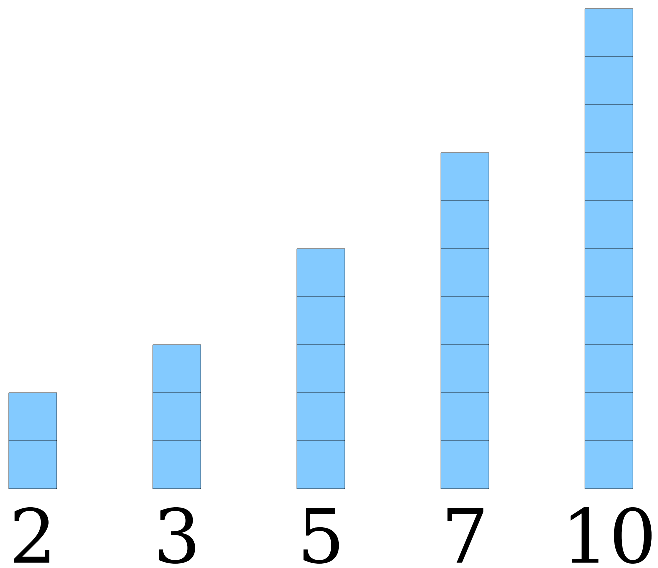
1	4	5	8	10	11	12	13

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

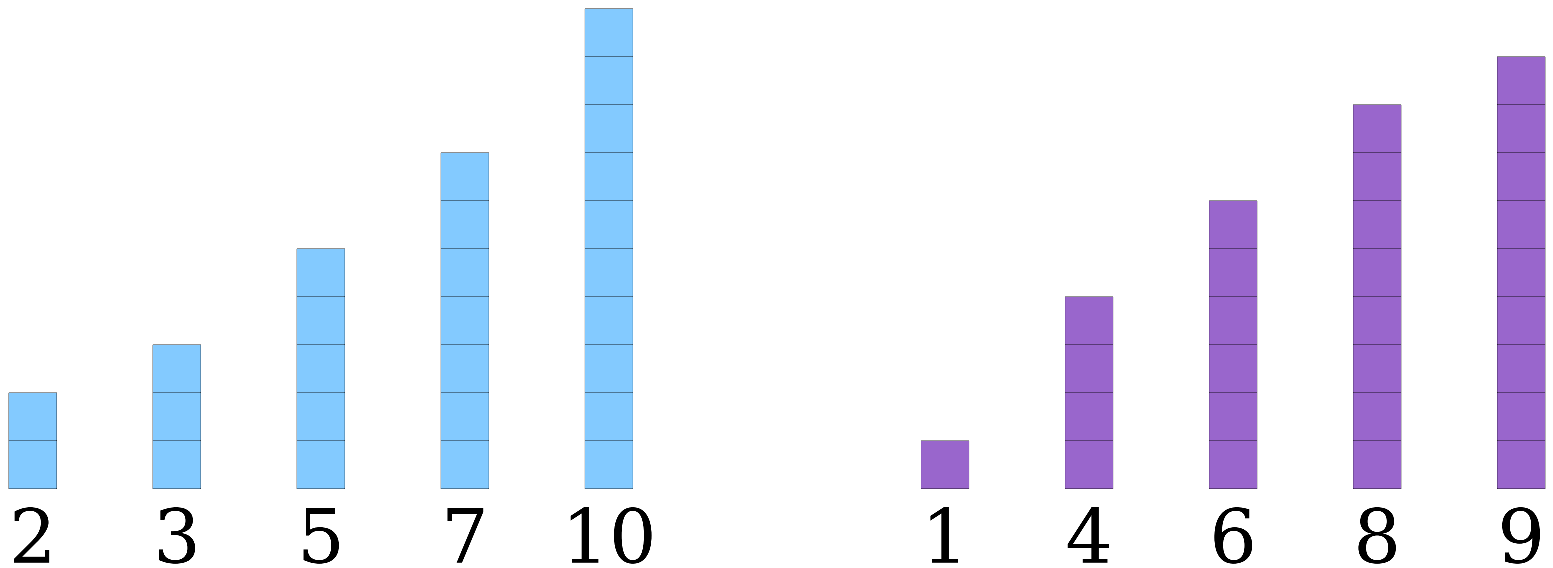
$$2 \cdot \tfrac{1}{4}T(n) = \tfrac{1}{2}T(n)$$

The Key Insight: ***Merge***

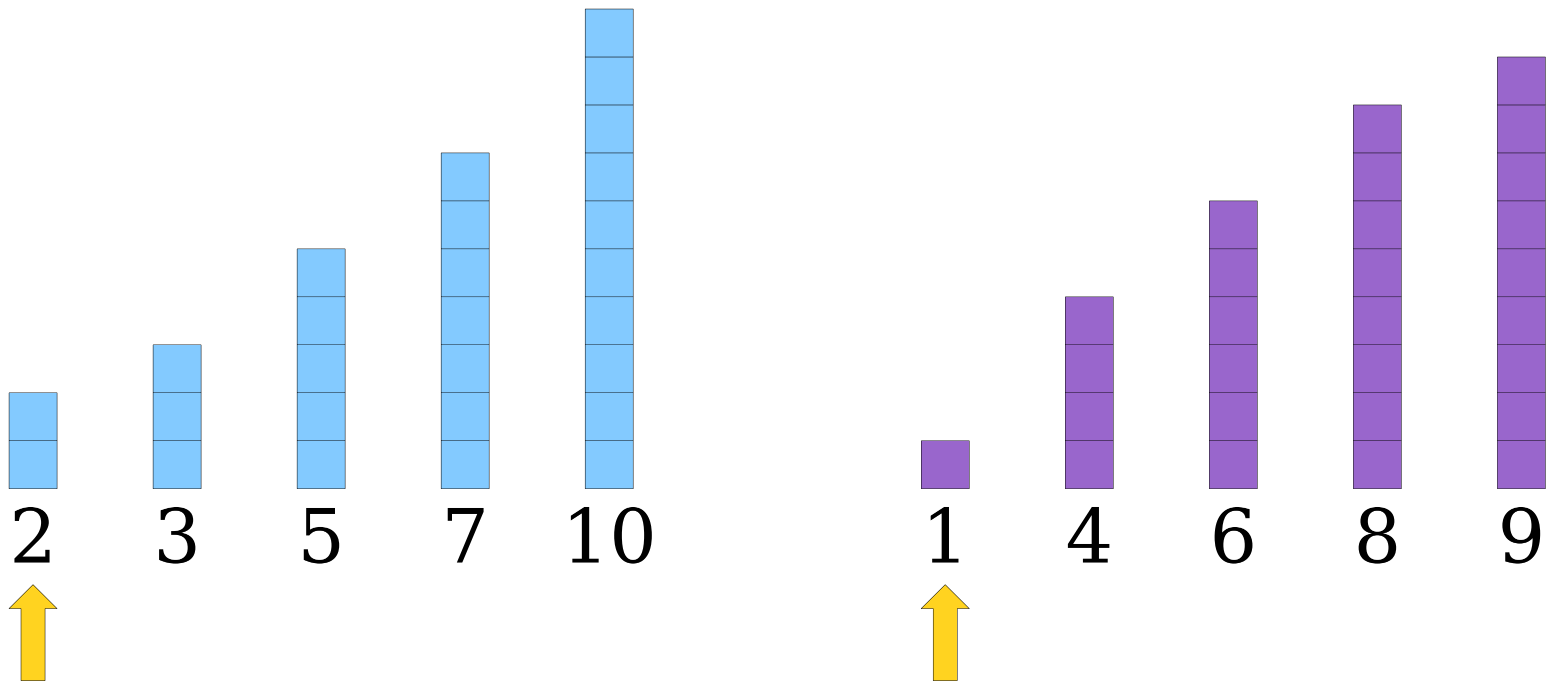
The Key Insight: ***Merge***



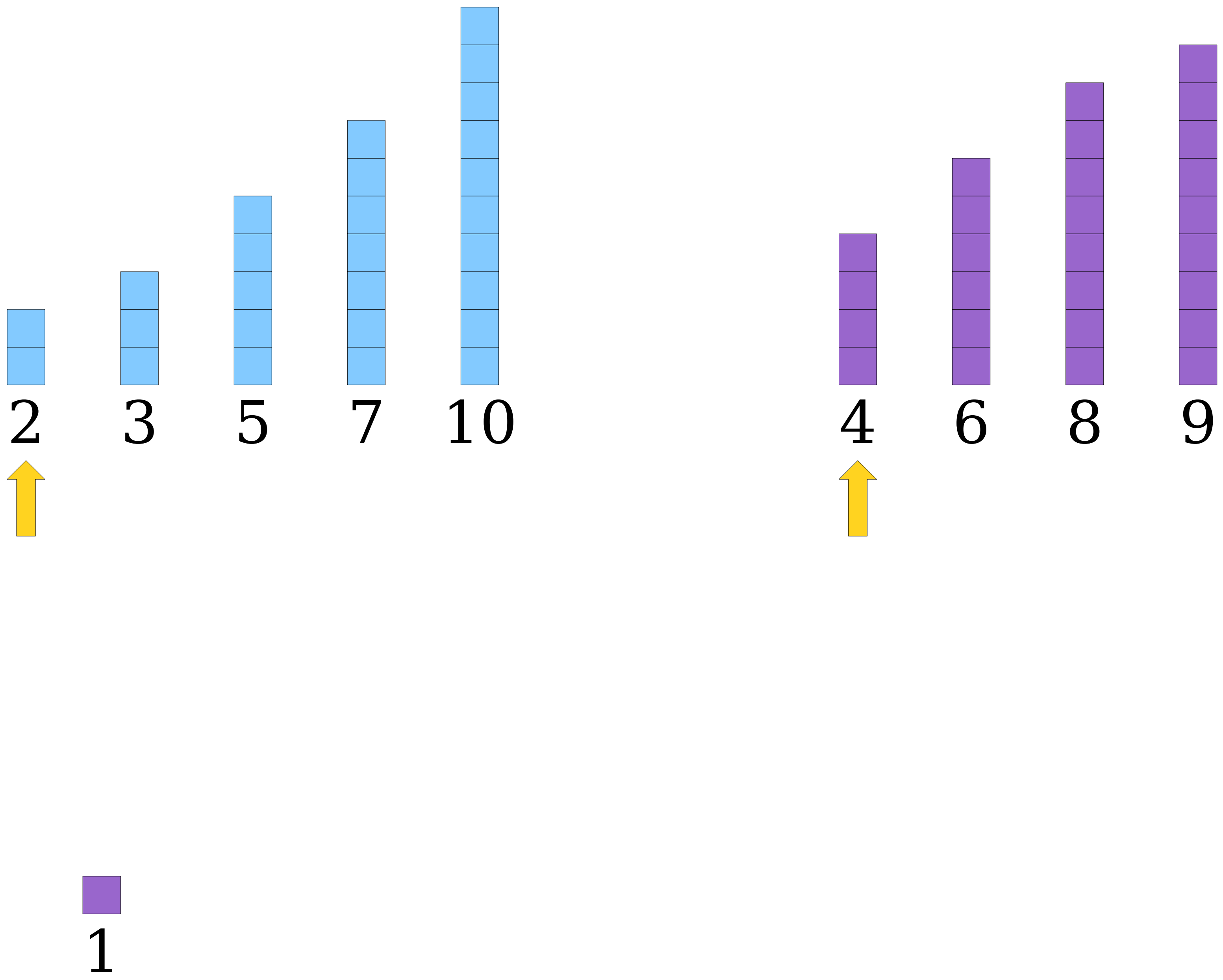
The Key Insight: ***Merge***



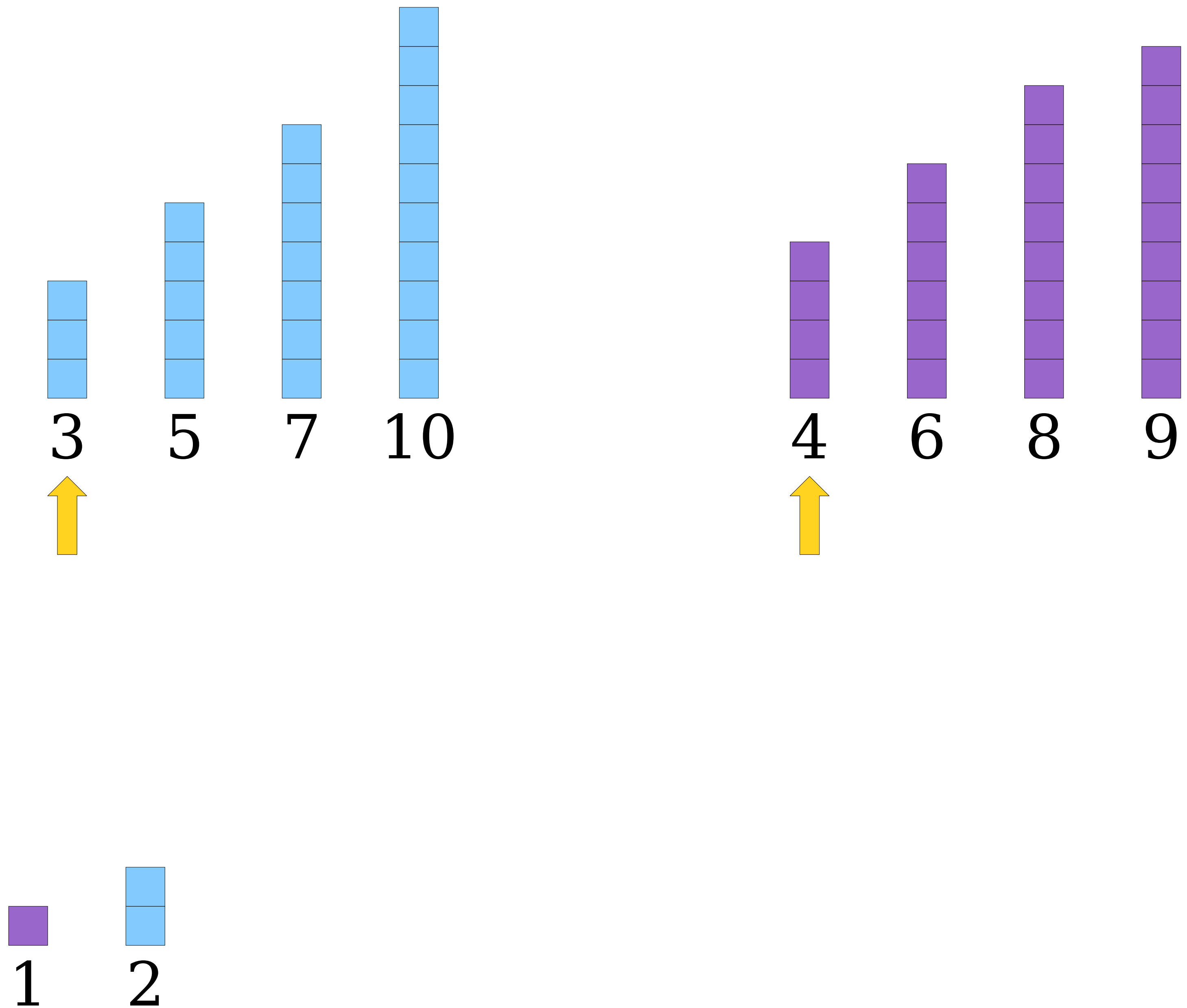
The Key Insight: ***Merge***



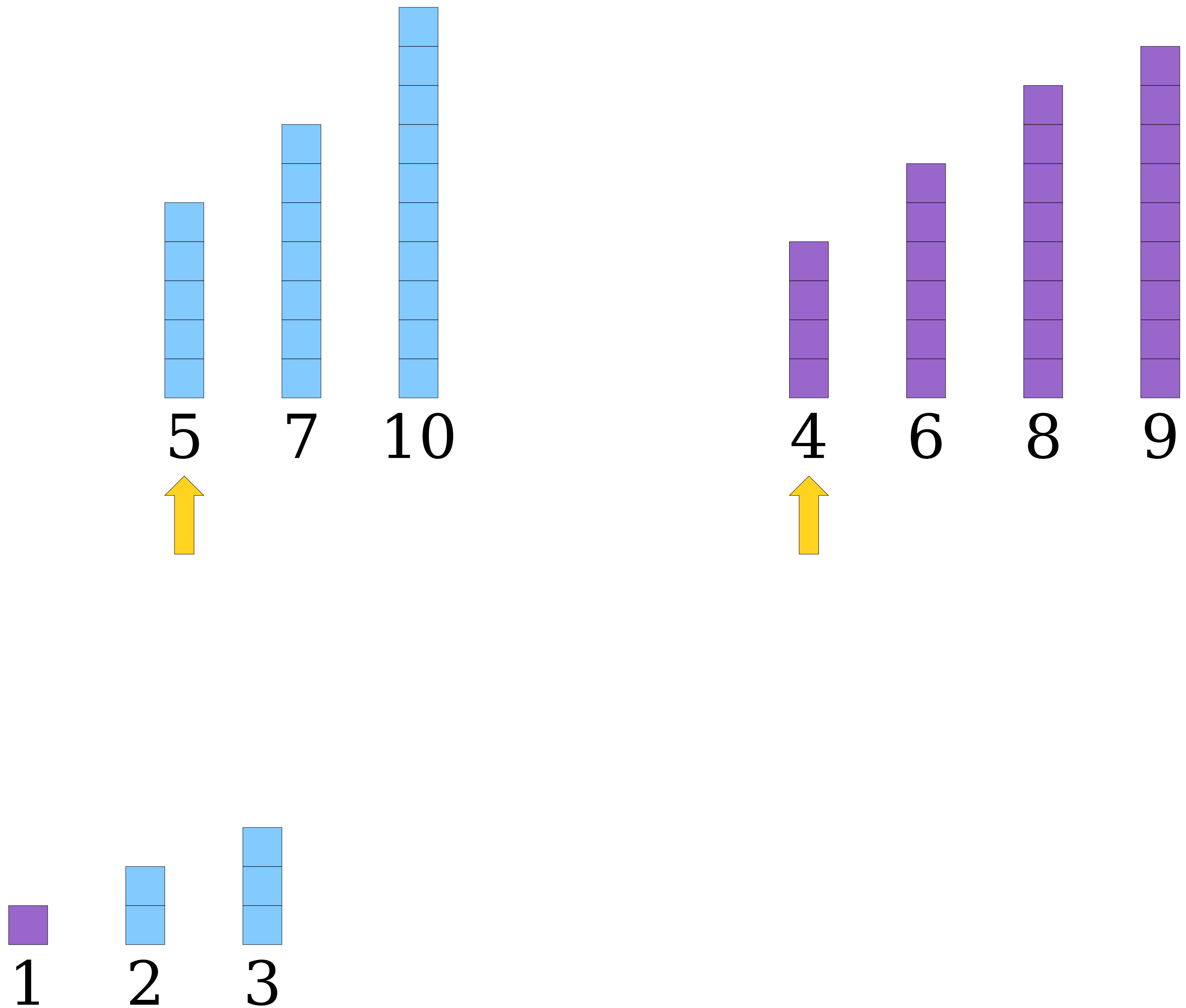
The Key Insight: ***Merge***



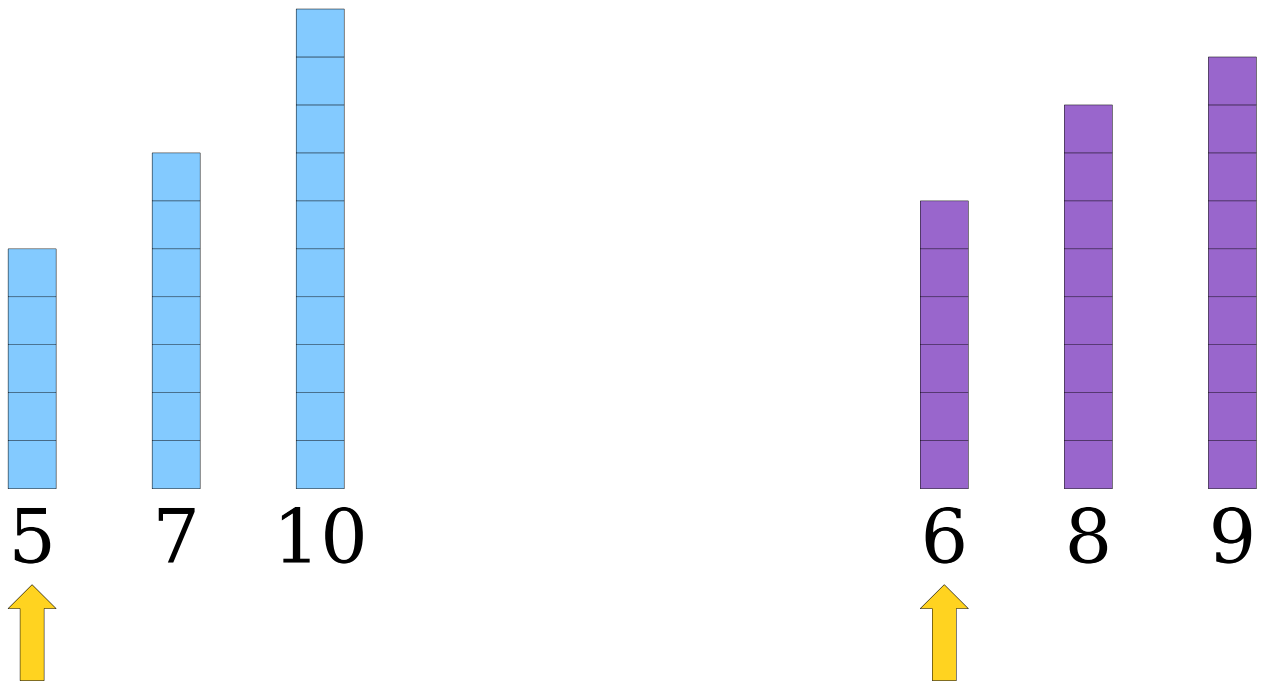
The Key Insight: ***Merge***

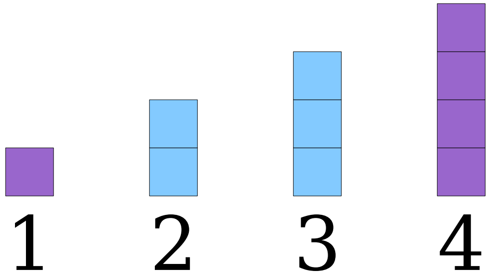


The Key Insight: ***Merge***

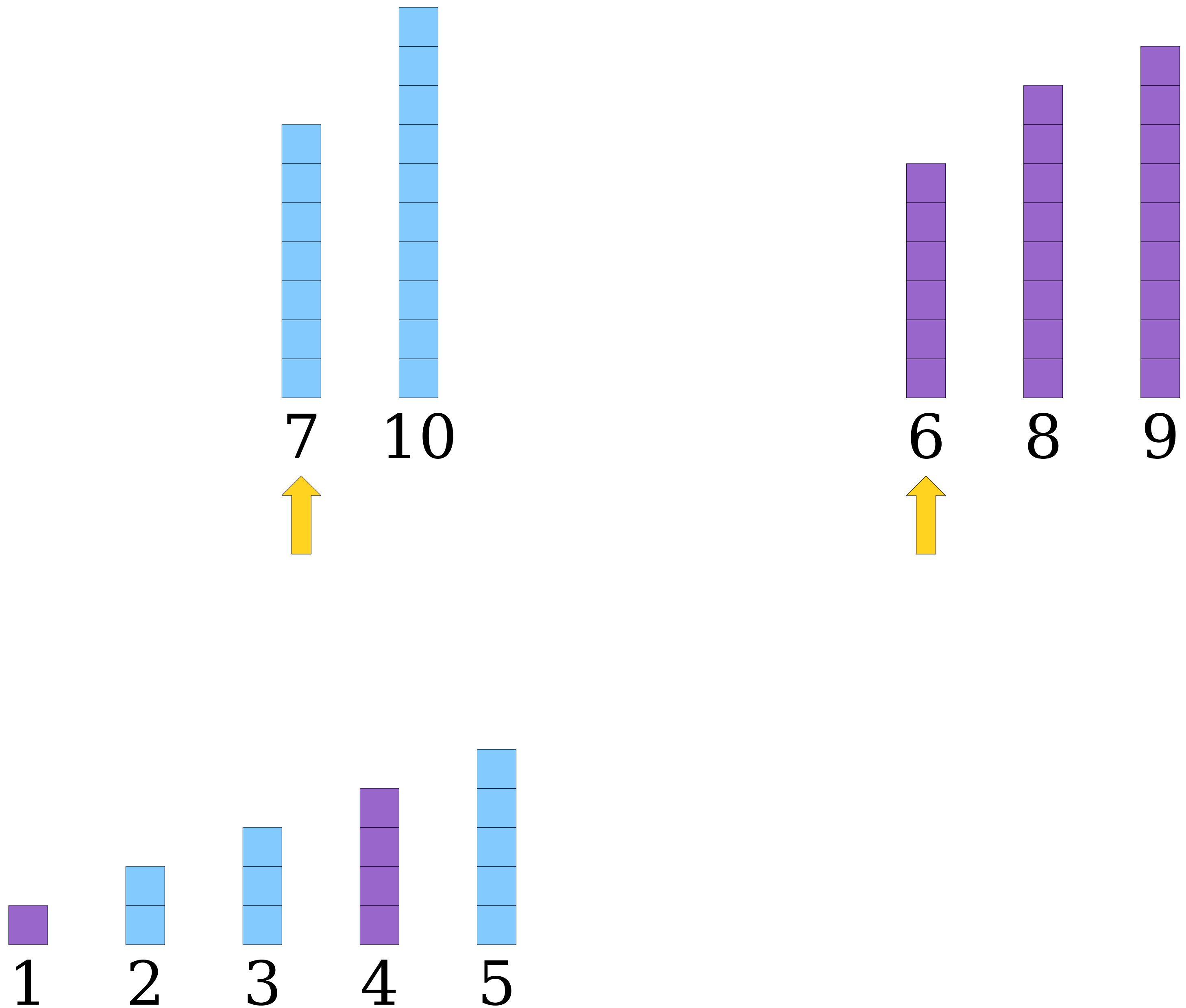


The Key Insight: ***Merge***

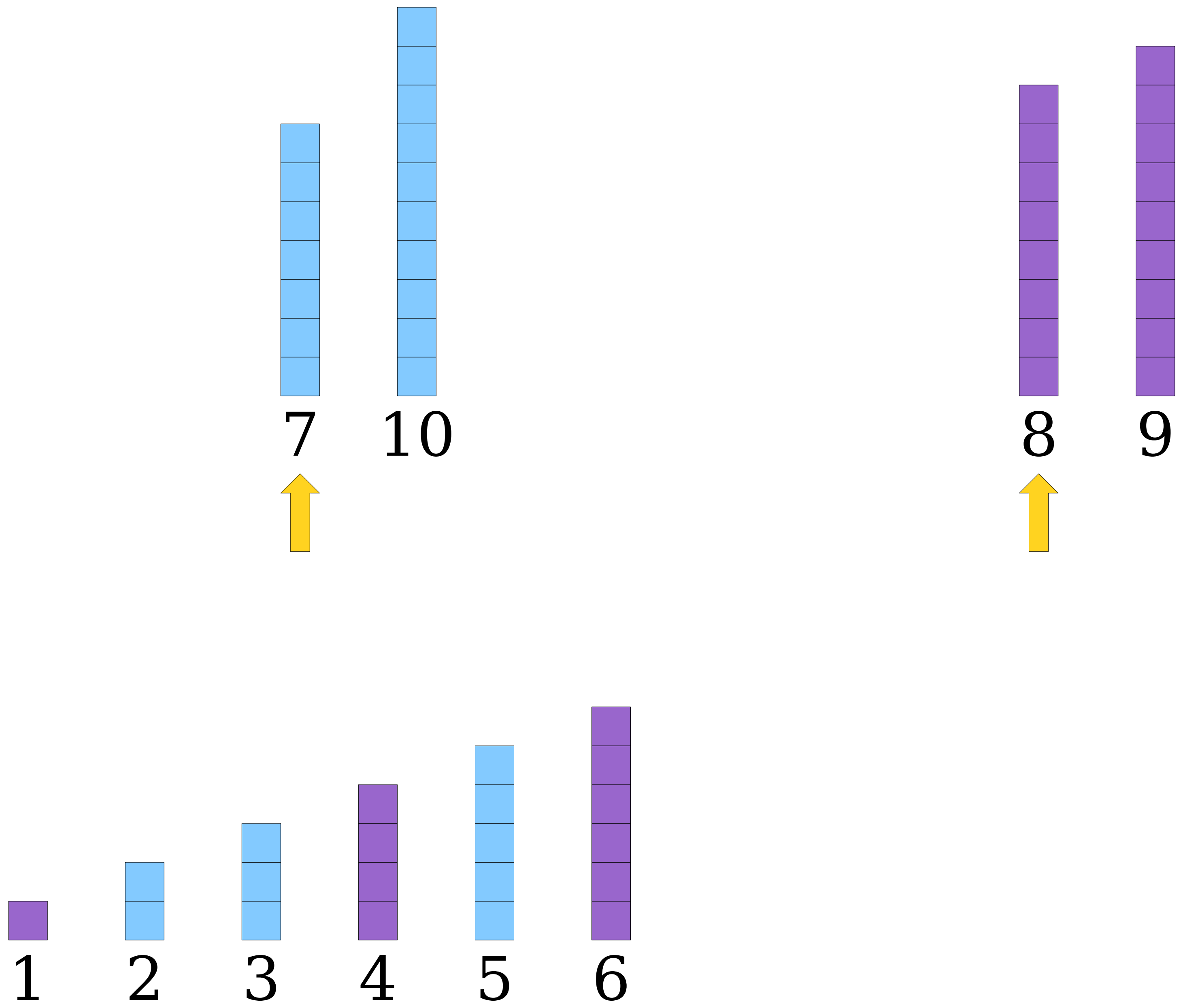




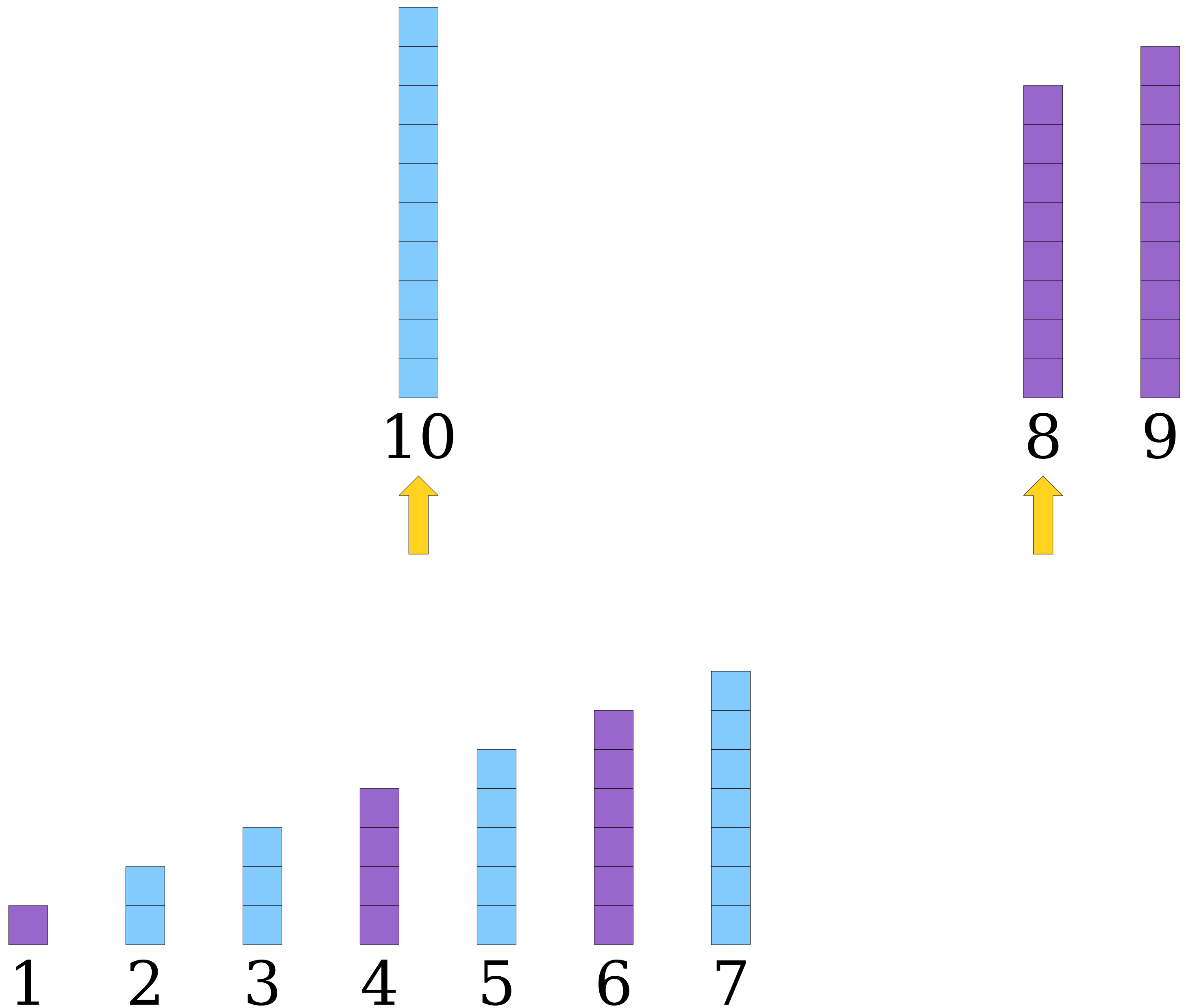
The Key Insight: ***Merge***



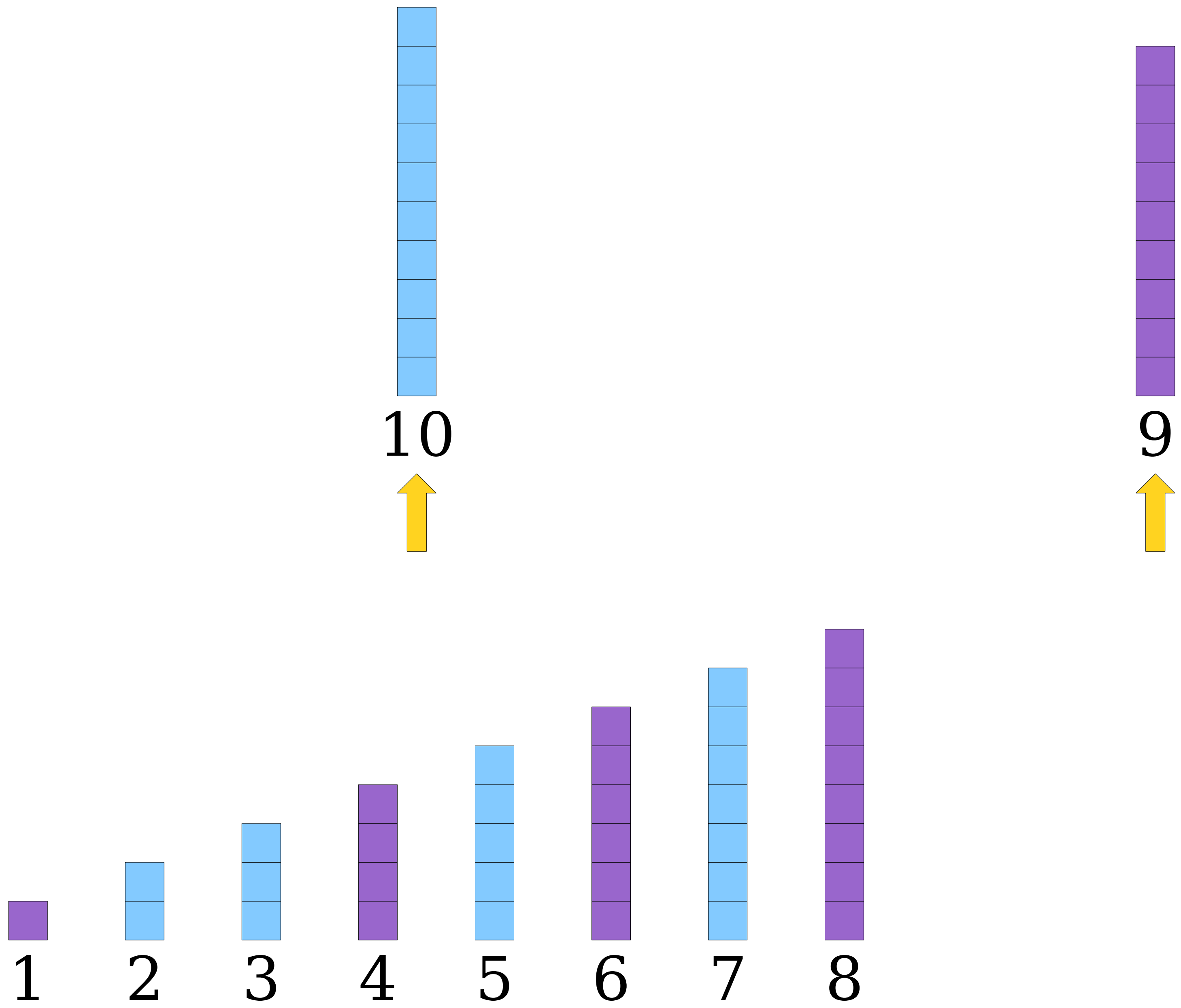
The Key Insight: ***Merge***



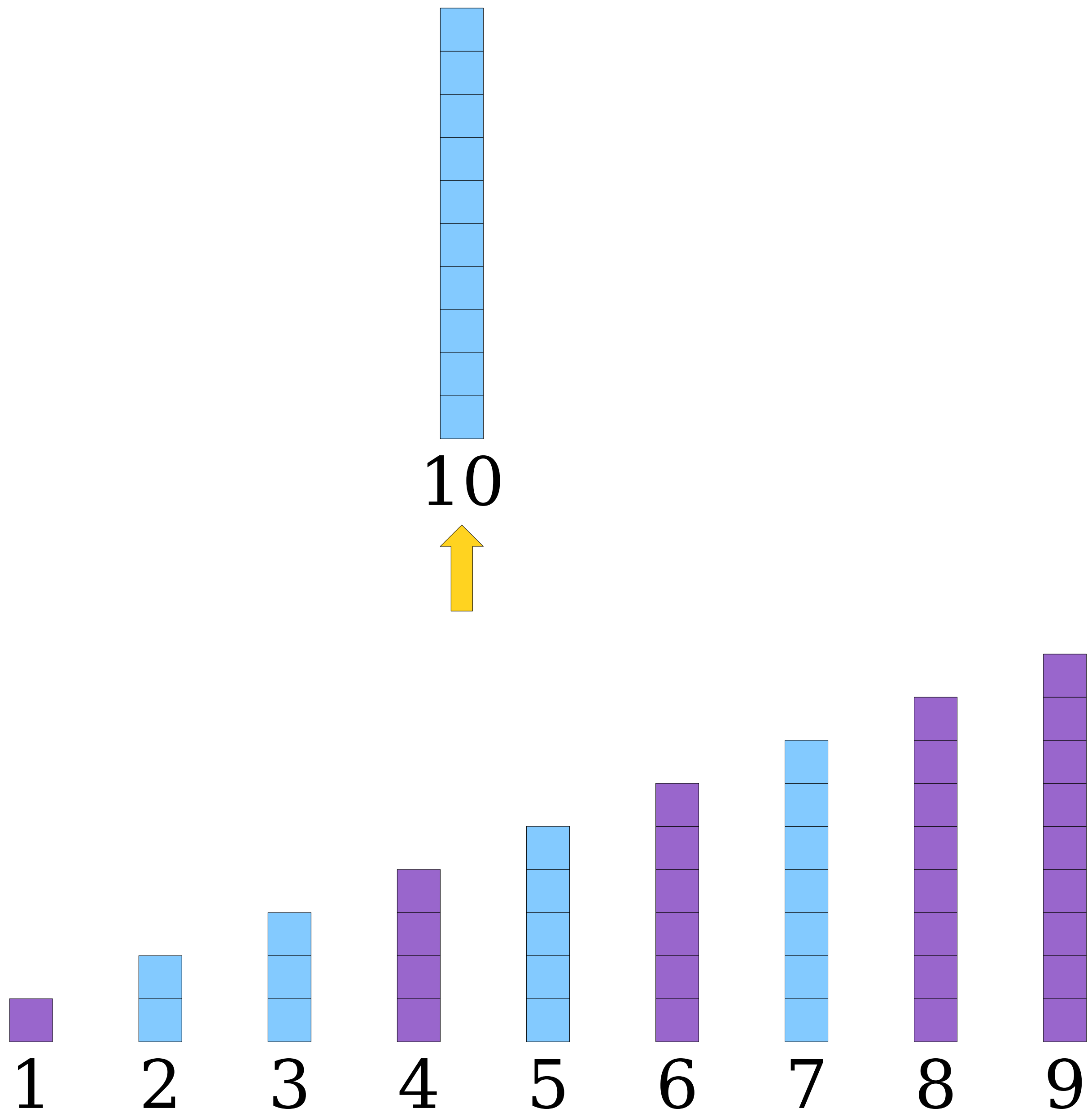
The Key Insight: ***Merge***



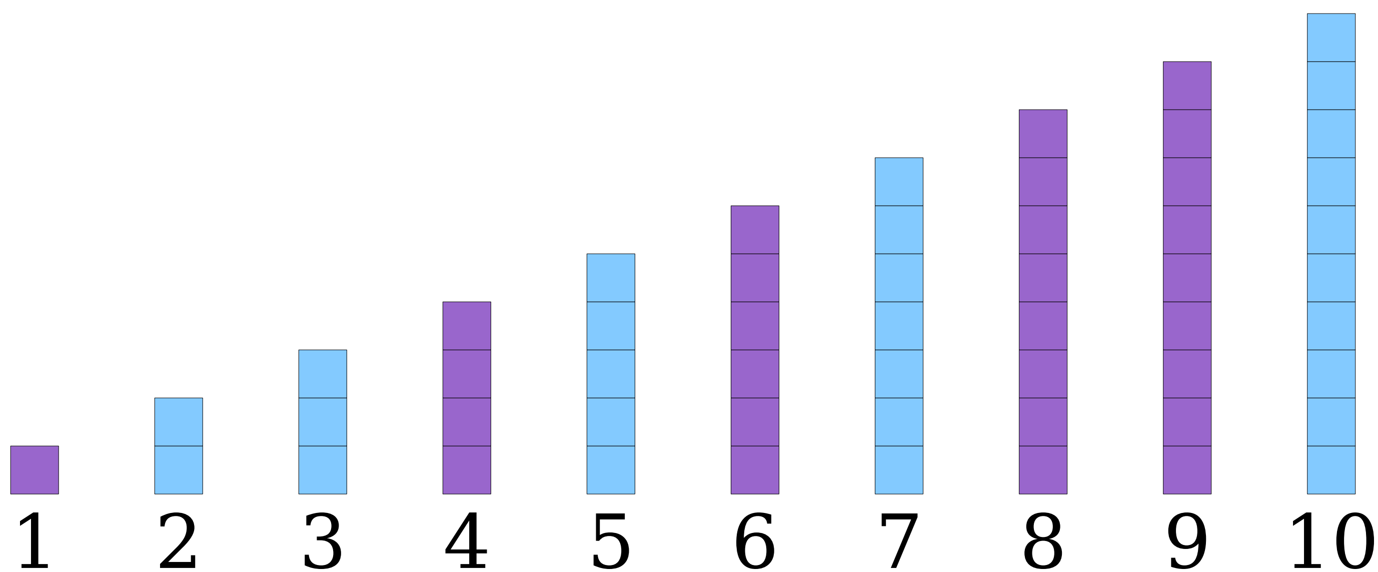
The Key Insight: ***Merge***



The Key Insight: ***Merge***



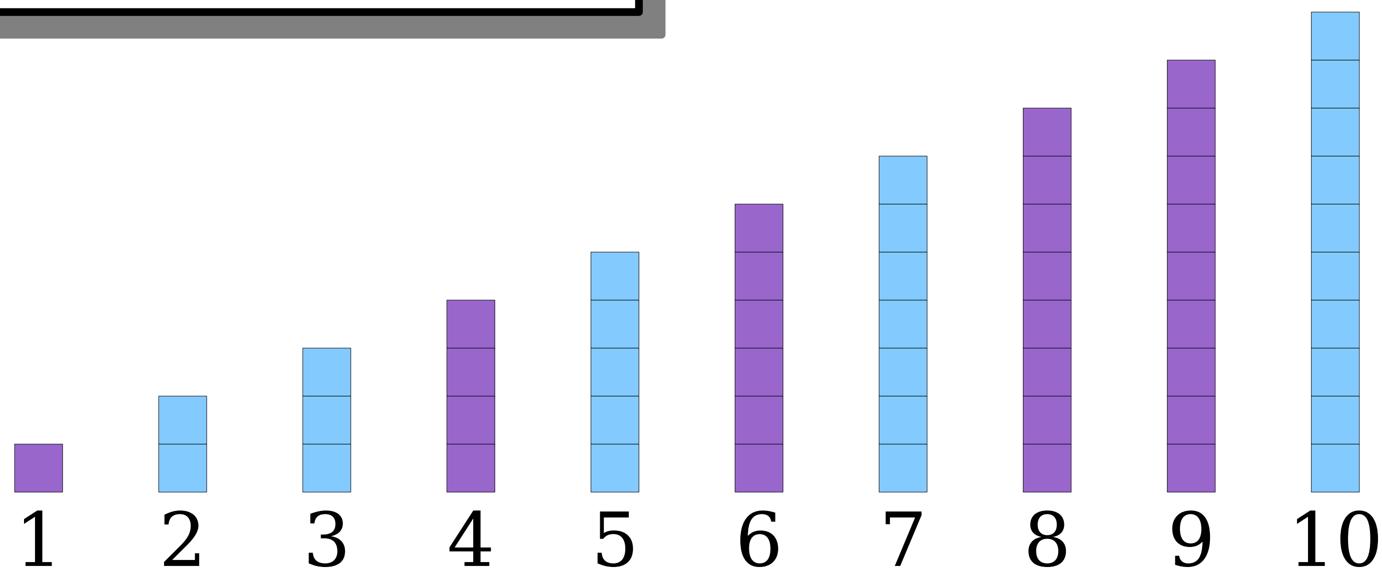
The Key Insight: ***Merge***



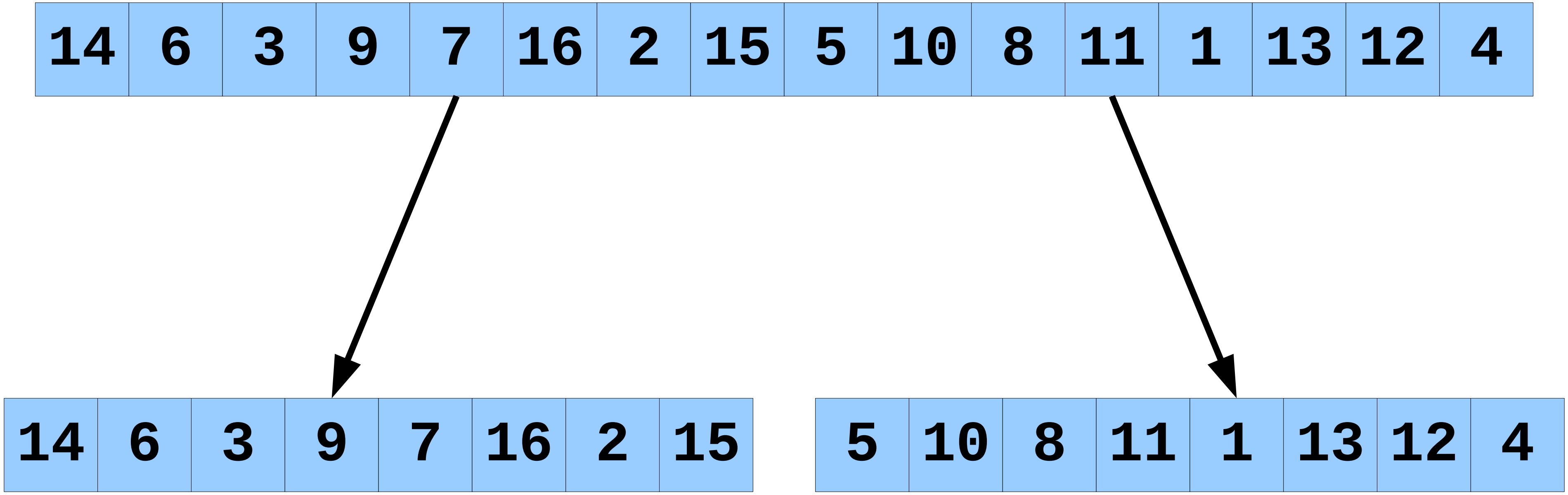
The Key Insight: ***Merge***

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

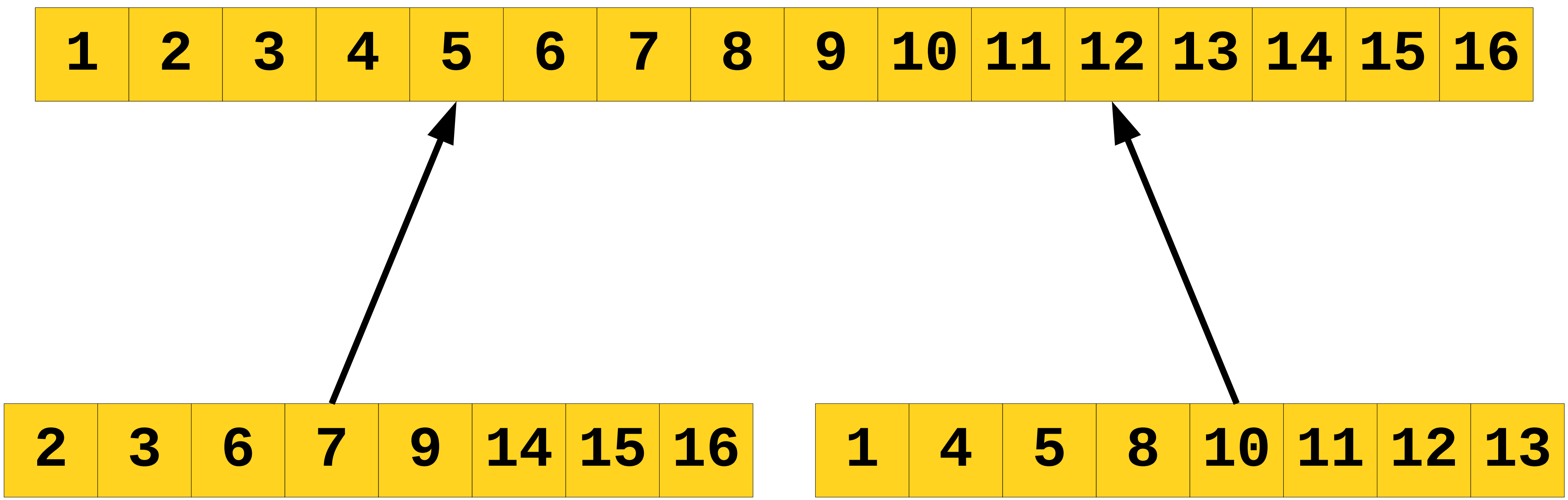
14	6	3	9	7	16	2	15

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”

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

$\mathrm{T}(n)$

14	6	3	9

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

7	16	2	15

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

5	10	8	11

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

1	13	12	4

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

“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

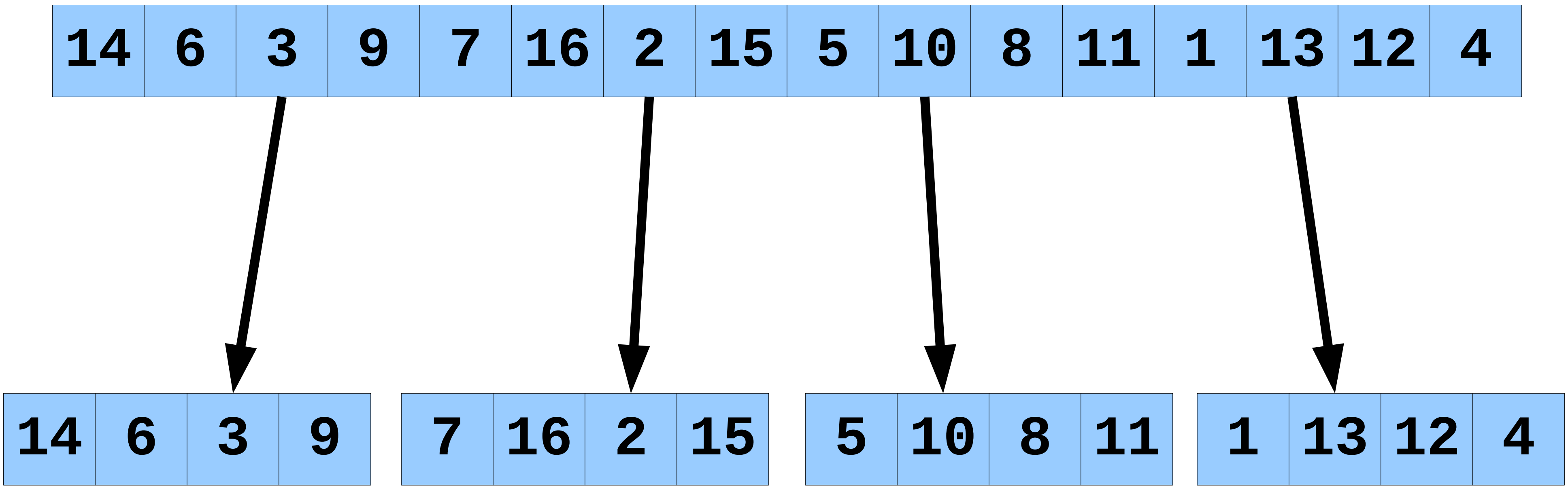
$^1/_{16}\ T(n)$

1	13	12	4

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

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

“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

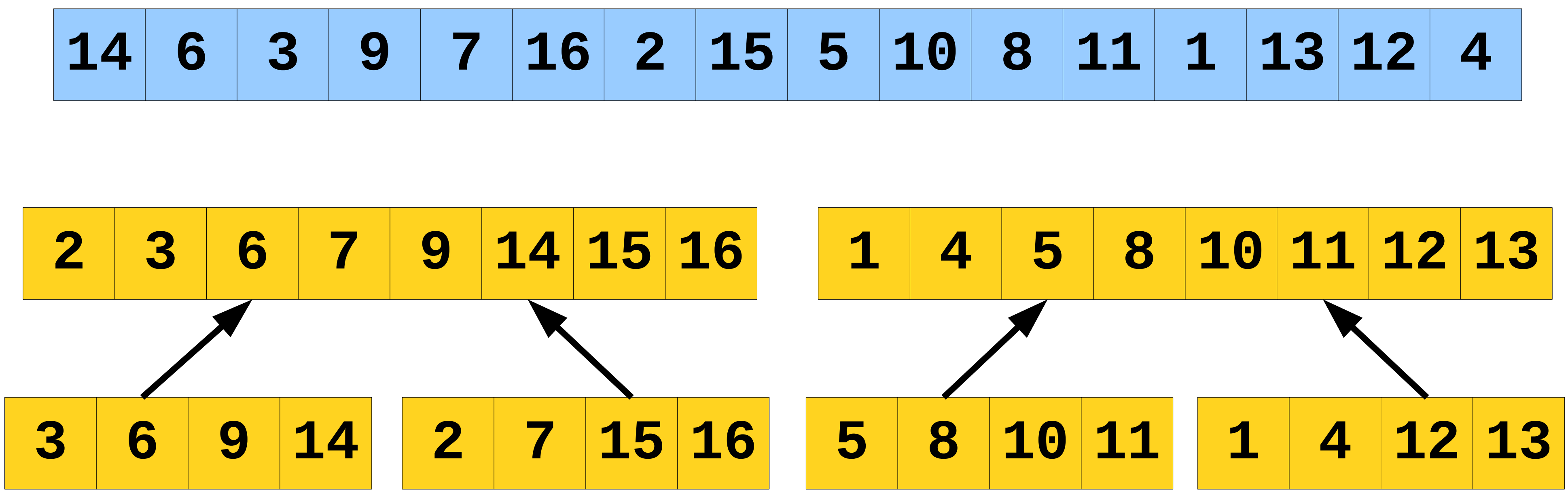
2	7	15	16

5	8	10	11

1	4	12	13

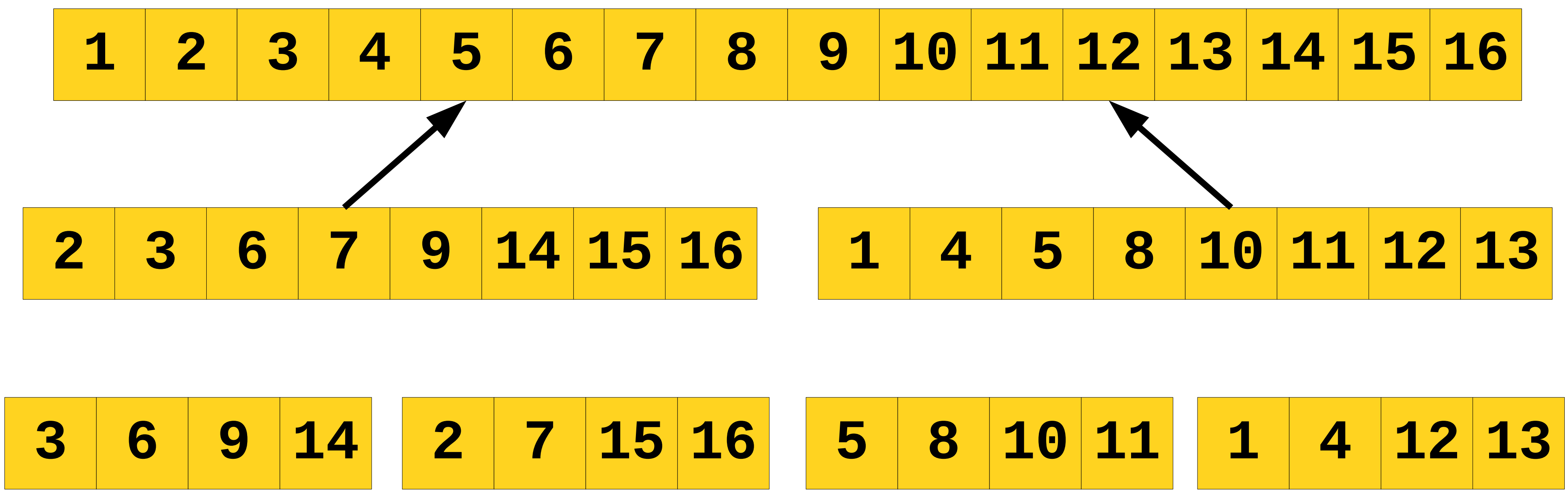
1. Split the input into quarters.
2. Insertion sort each quarter.

“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”

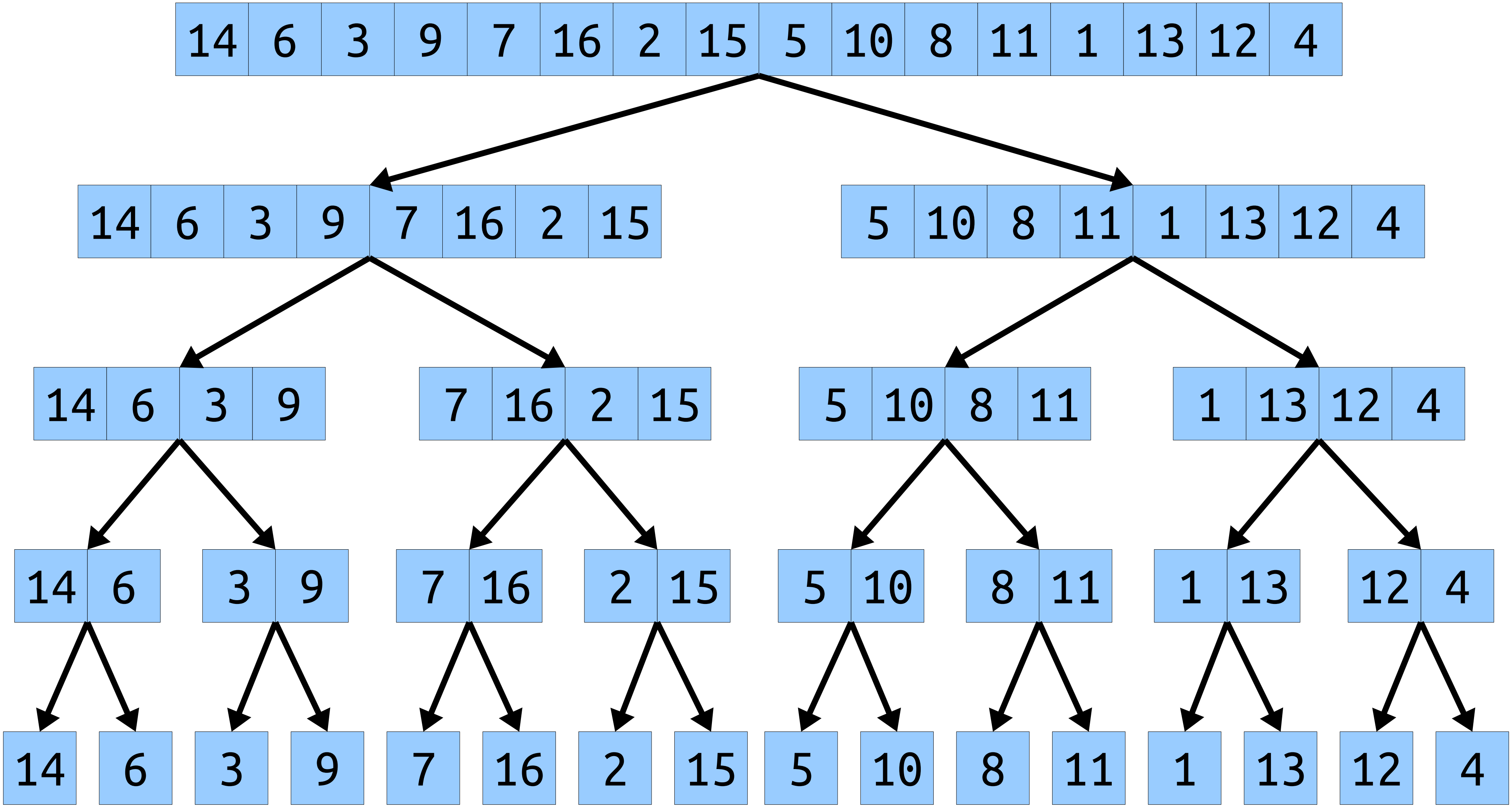


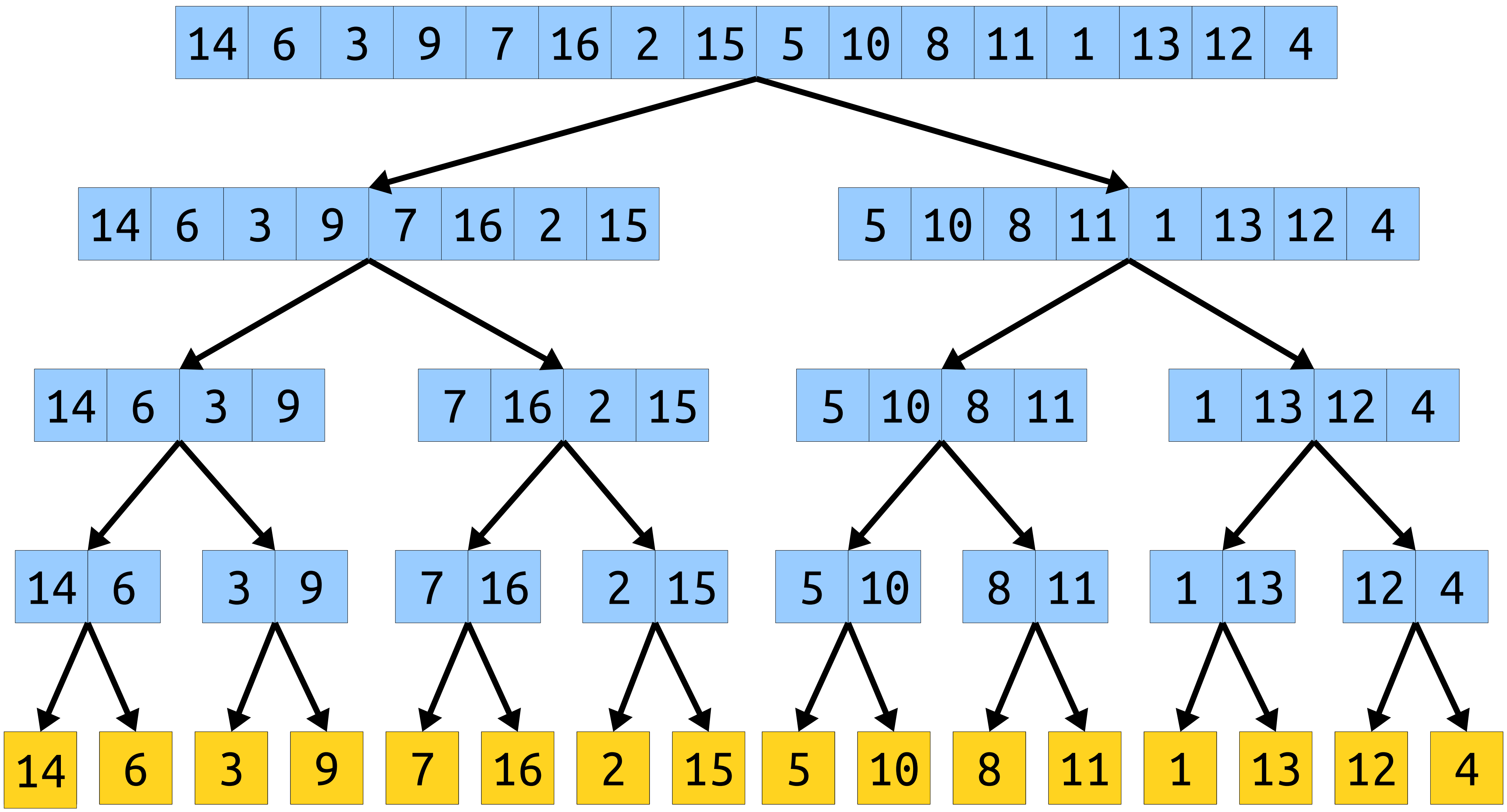
1. Split the input into quarters.
2. Insertion sort each quarter.
3. Merge two pairs of quarters into halves.
4. Merge the two halves back together.

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

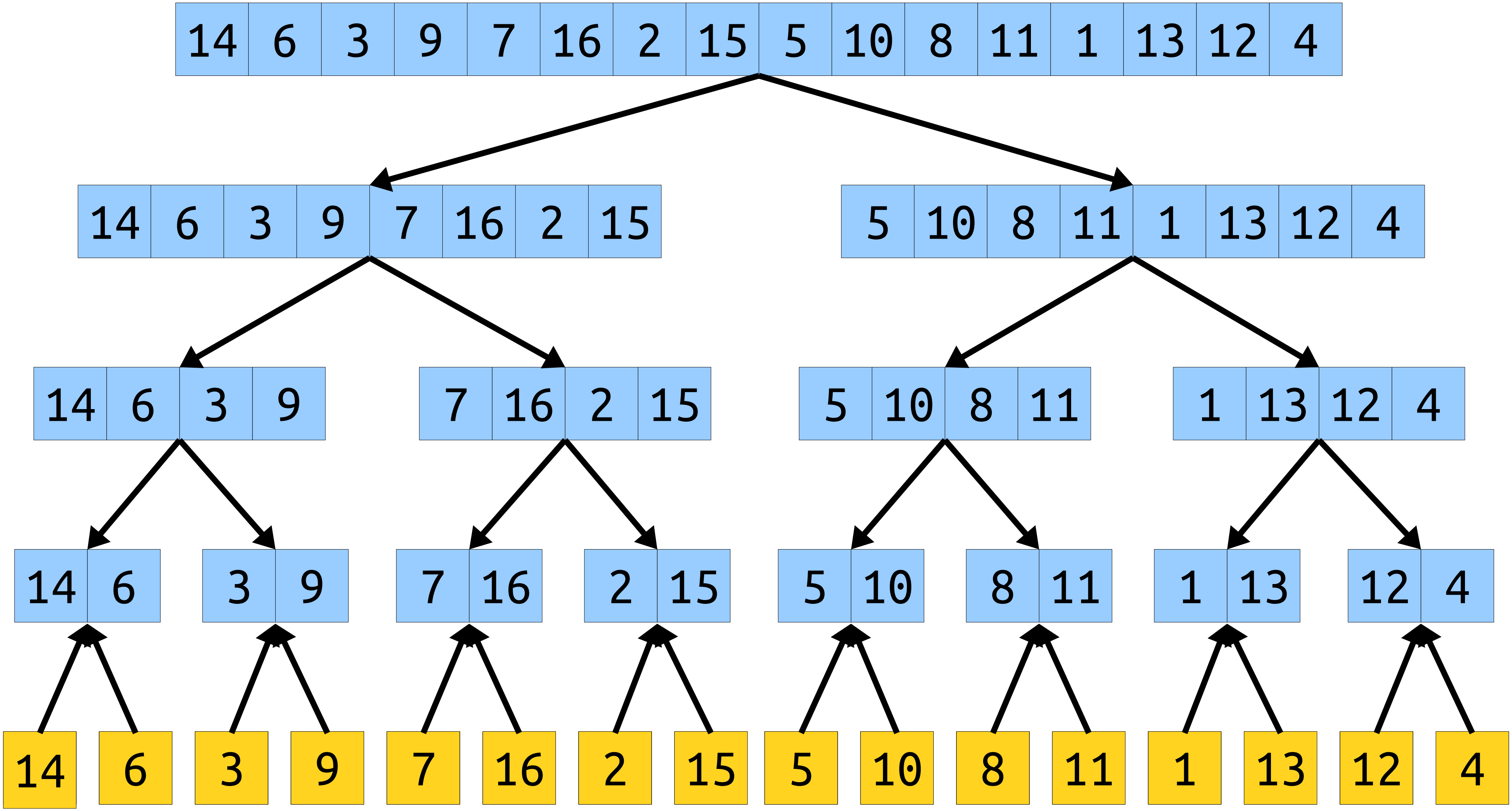
Splitting to the Extreme

- 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?*

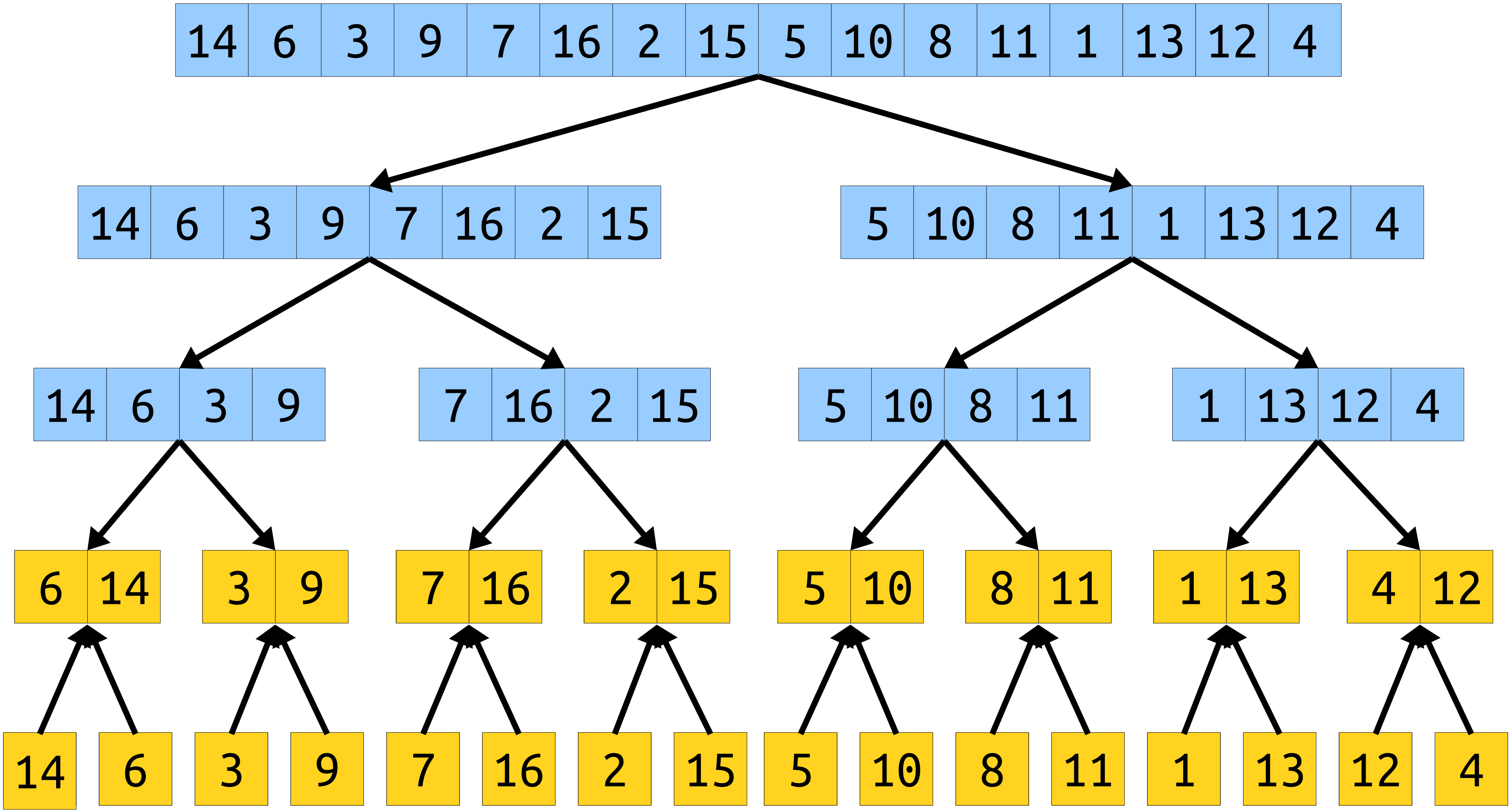

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

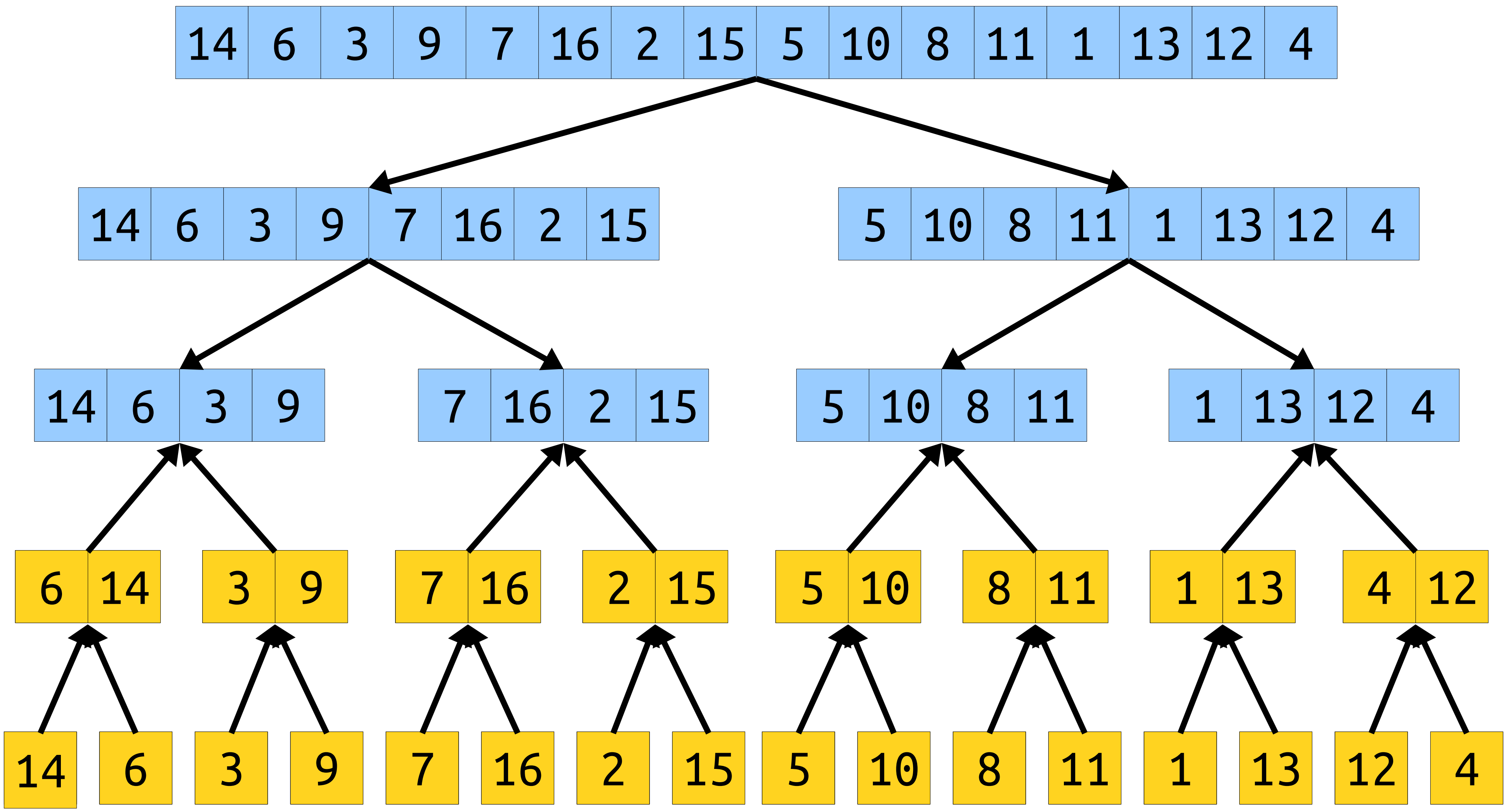


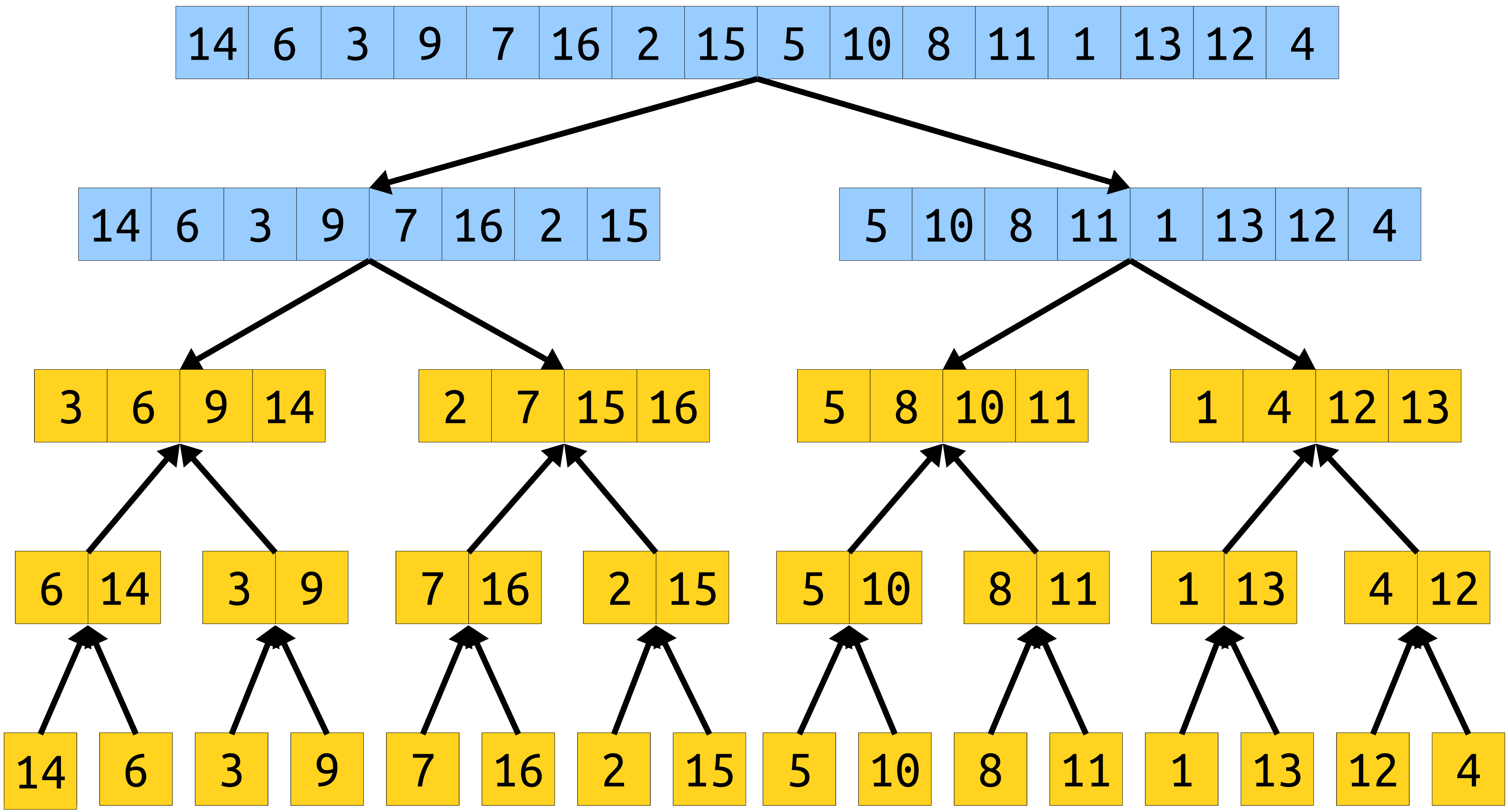
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

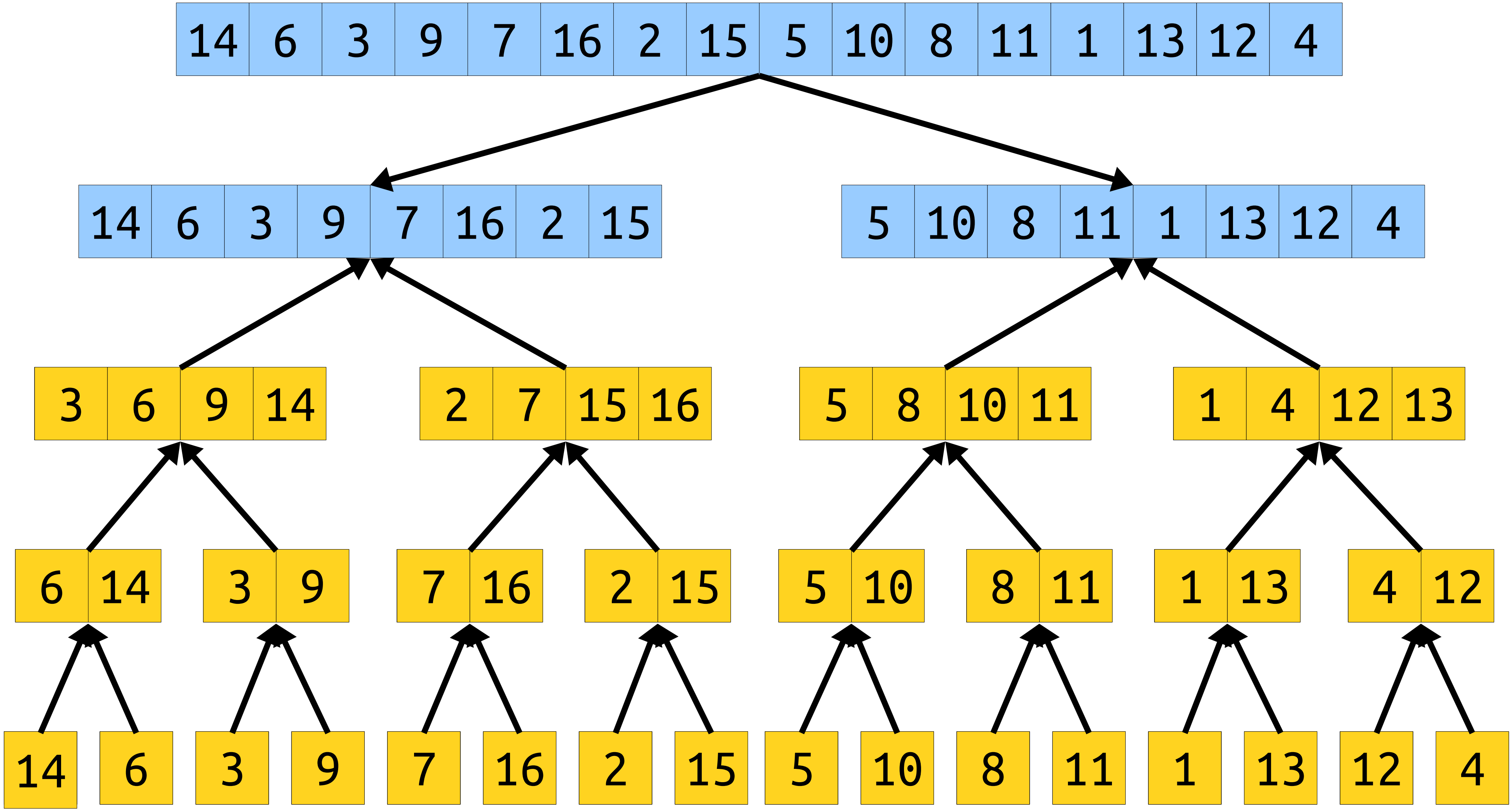


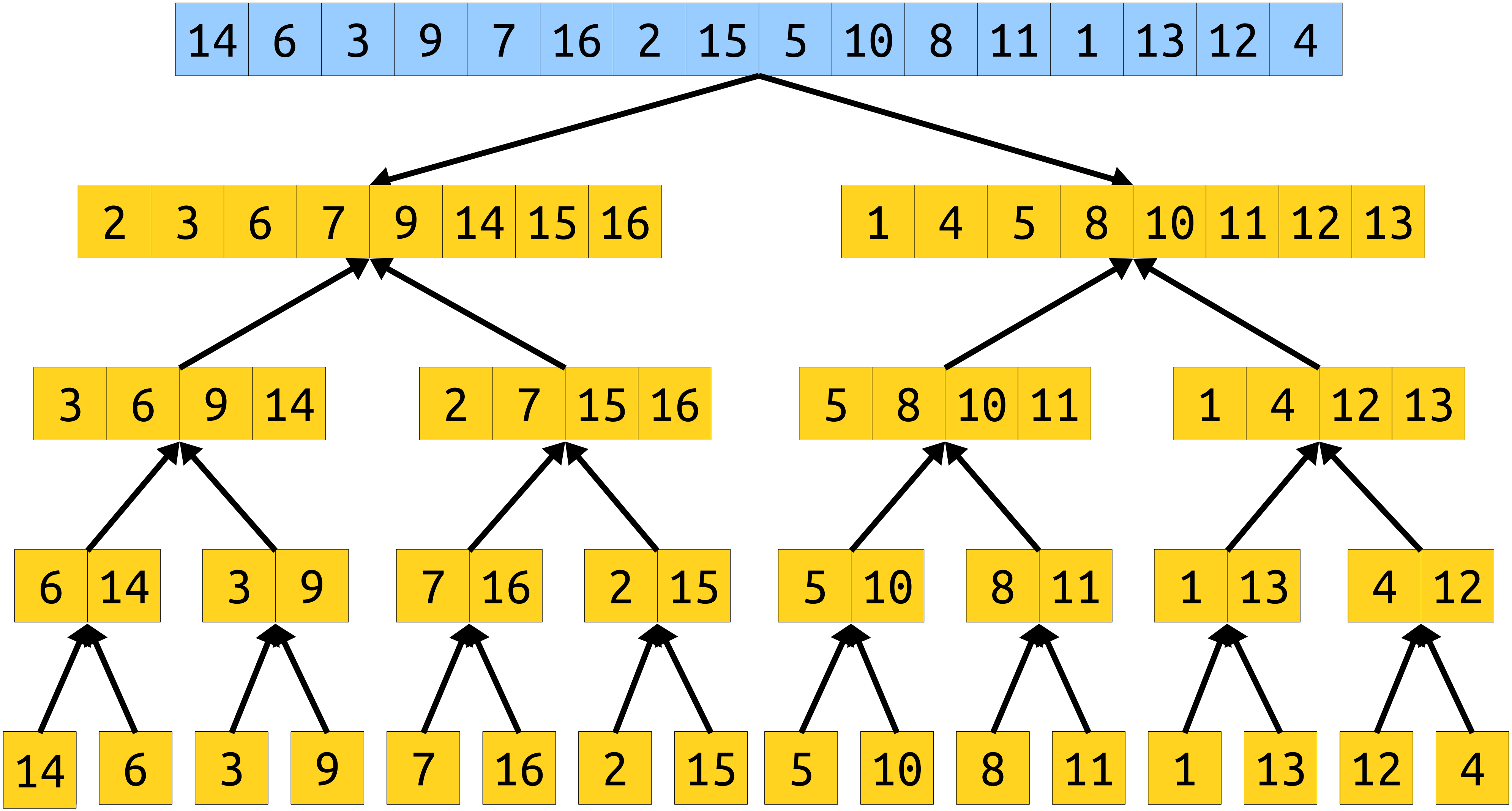
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4

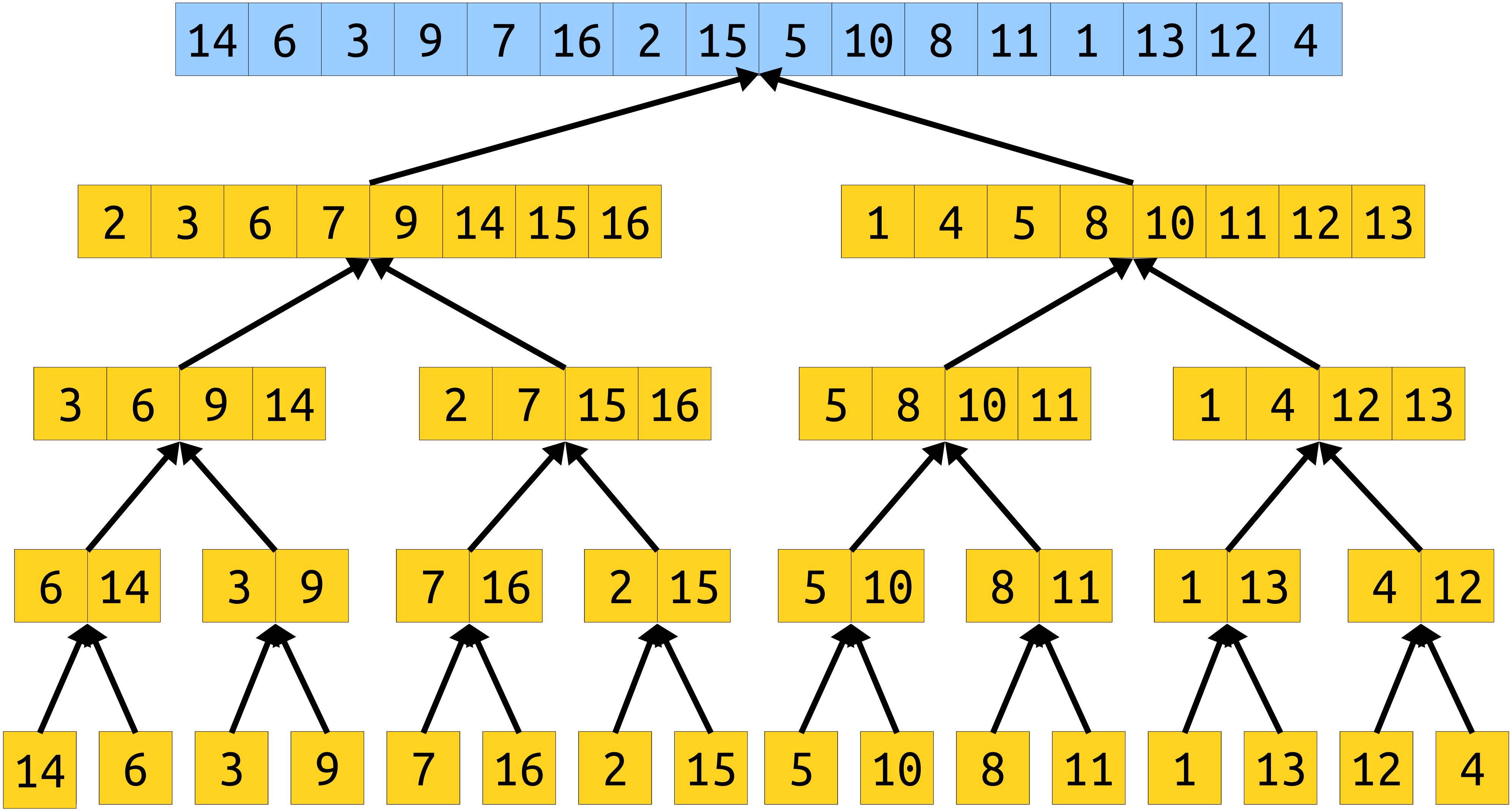

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4


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

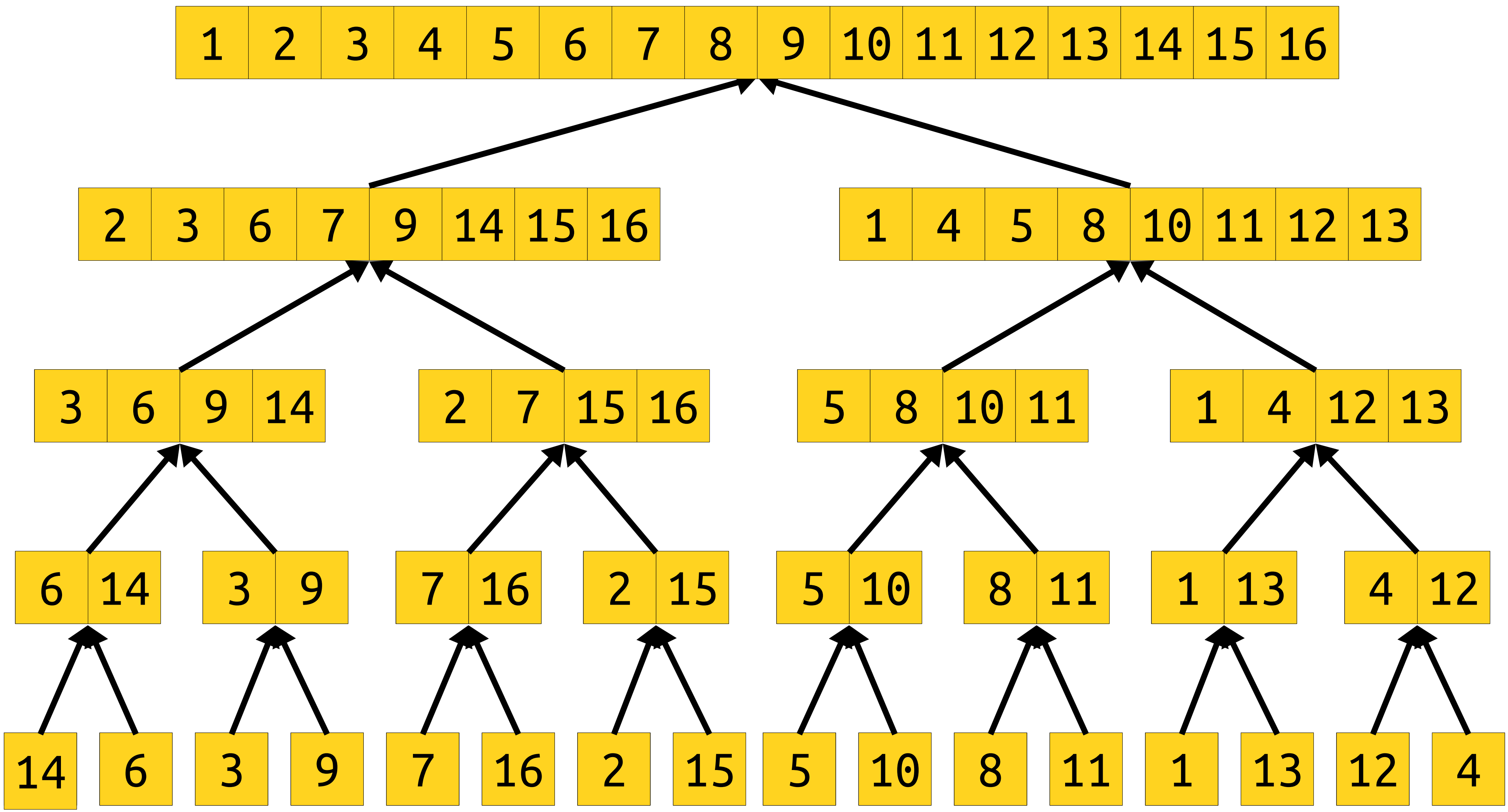

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4


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

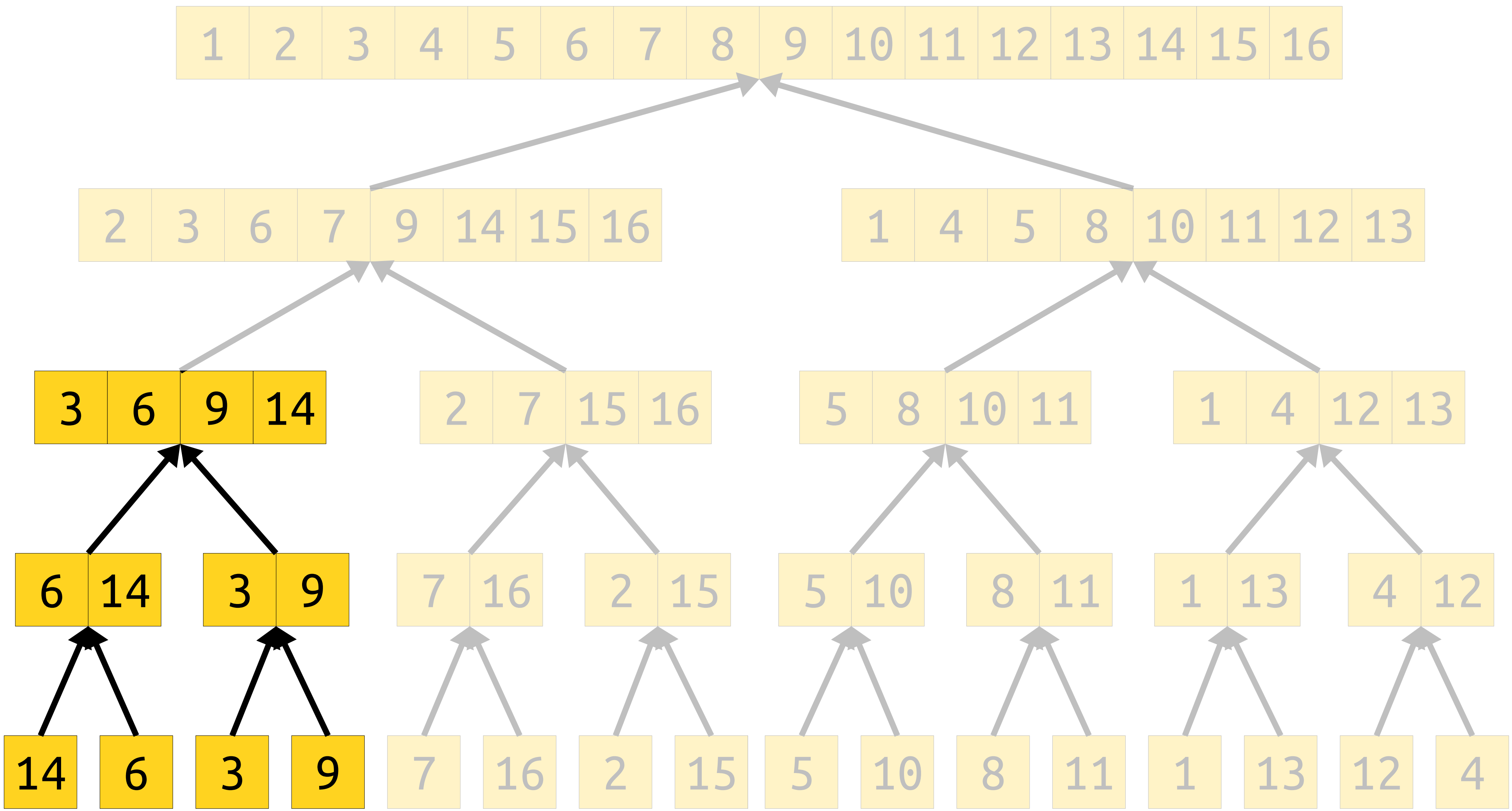

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
2 3 6 7 9 14 15 16
1 4 5 8 10 11 12 13
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4



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

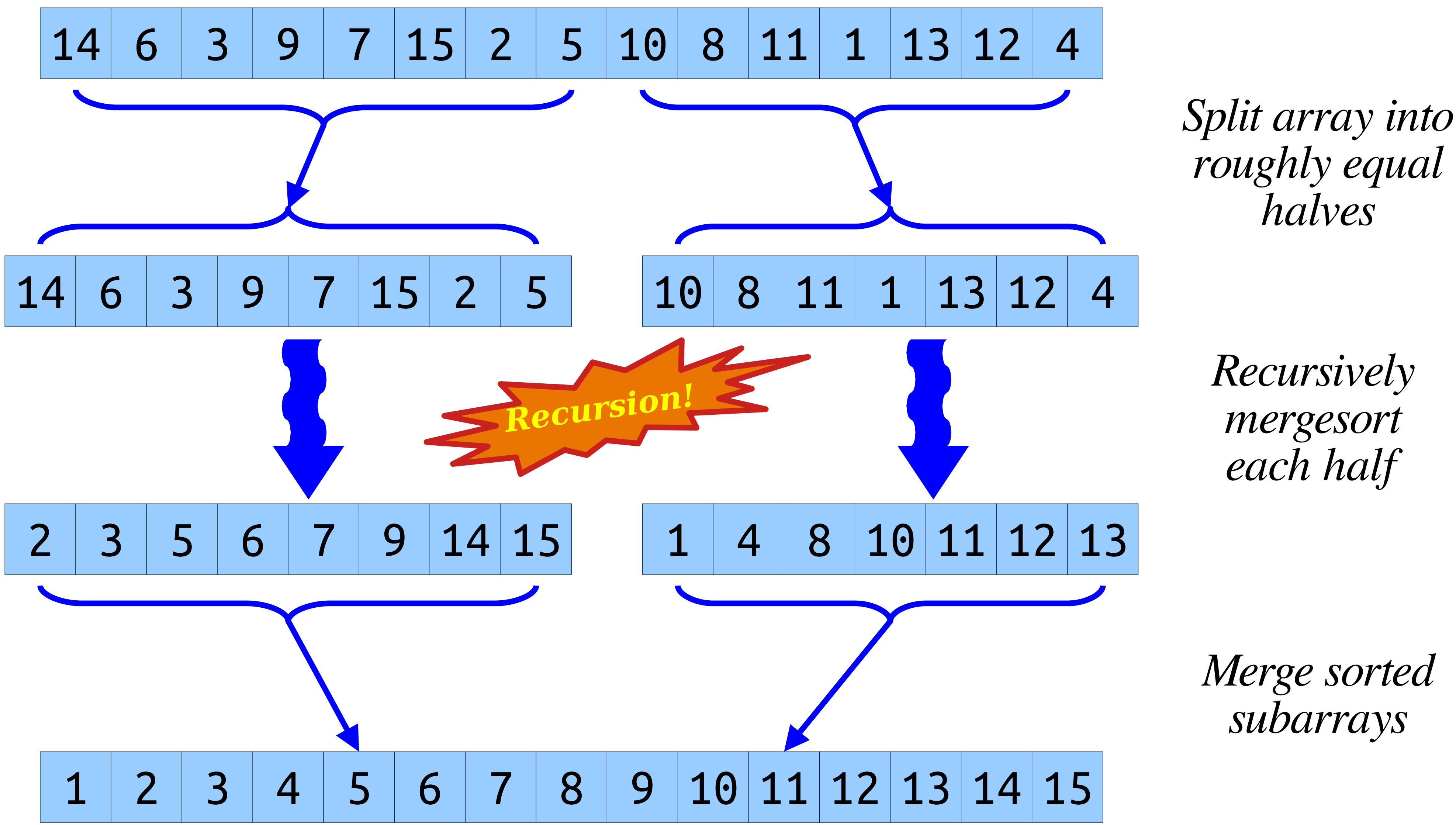


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
2 3 6 7 9 14 15 16
1 4 5 8 10 11 12 13
3 6 9 14
2 7 15 16
5 8 10 11
1 4 12 13
6 14
3 9
7 16
2 15
5 10
8 11
1 13
4 12
14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4



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



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!

```
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);
}
```

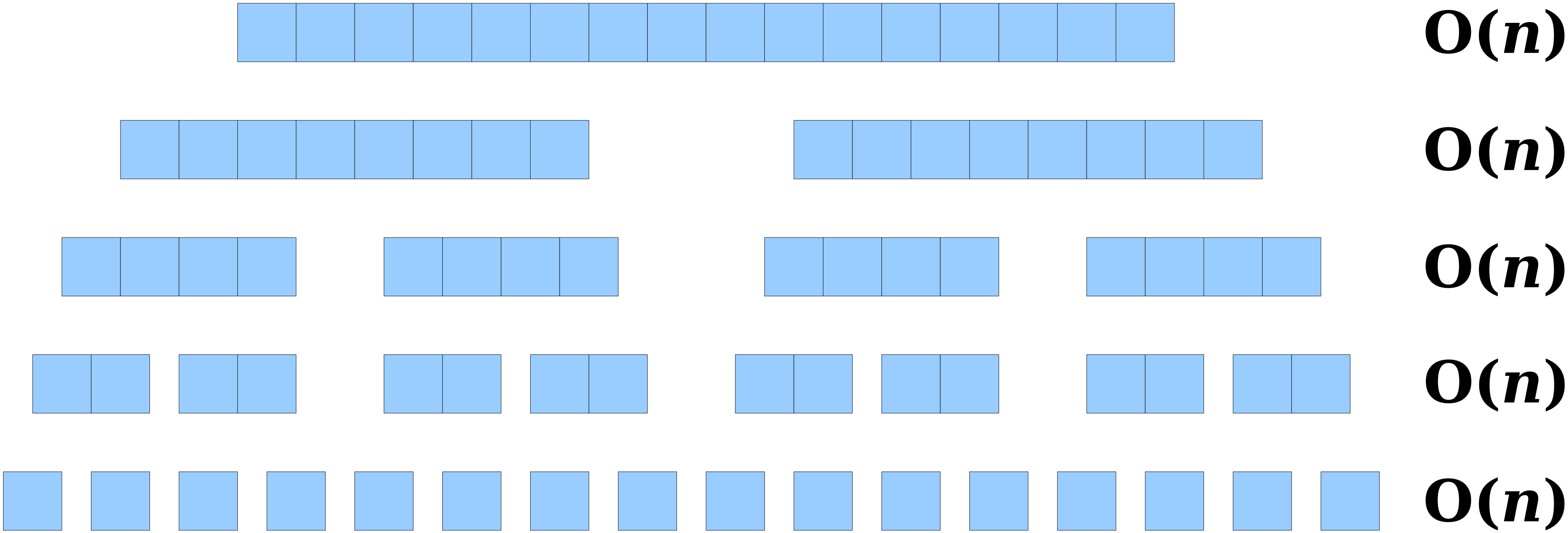
```
void mergesort(Vector<int>& v) {
   /* Base case: 0- or 1-element lists are
    * already sorted.
    */
   if (v.size() <= 1) {
      return;
   }

   /* 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

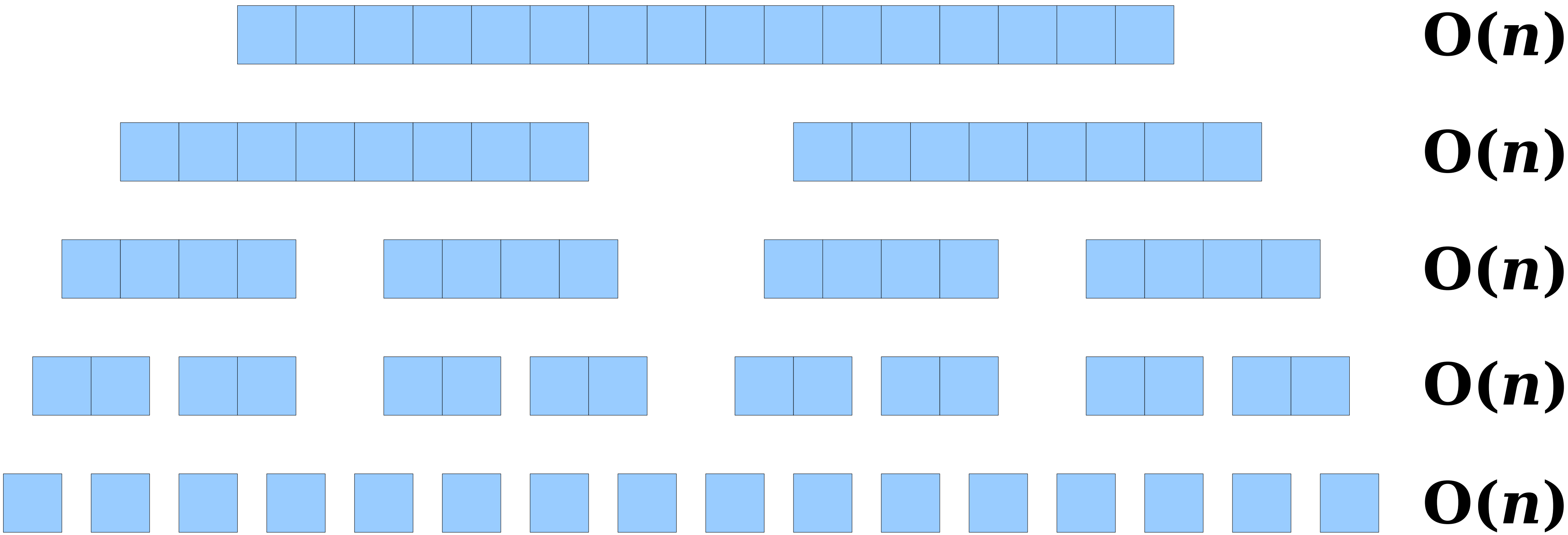
```
   /* Recursively sort these arrays. */
   mergesort(left);
   mergesort(right);

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

$O(n)$ work

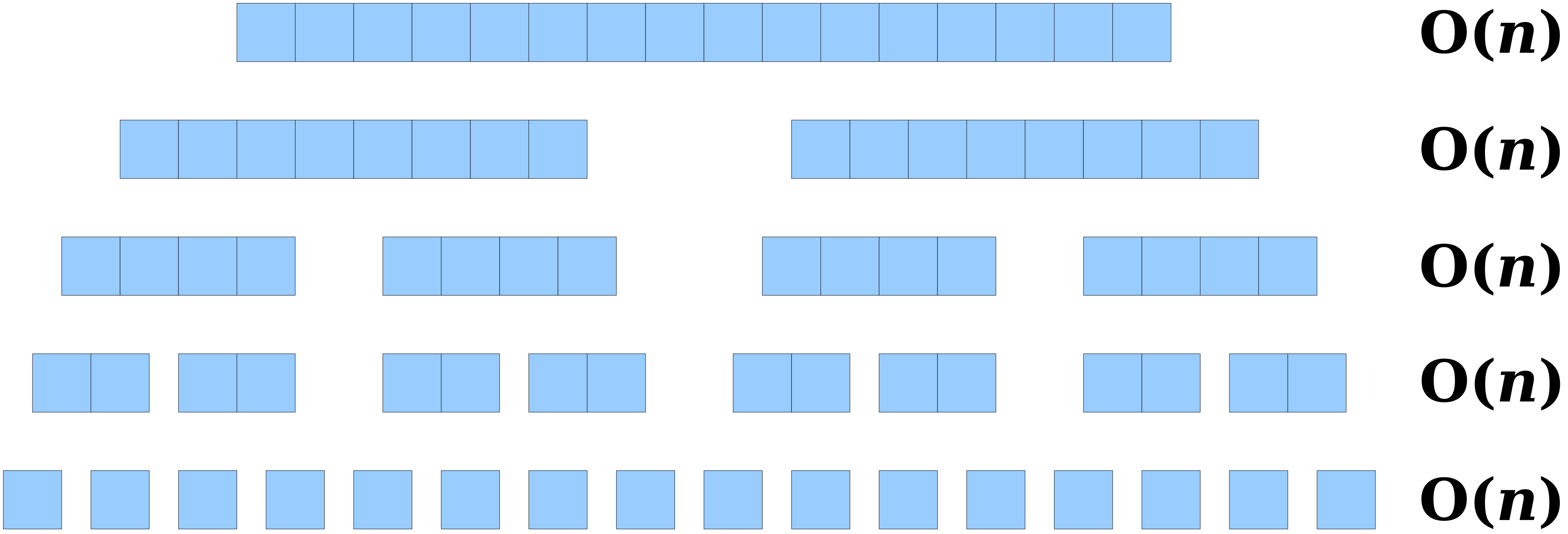
```
void mergesort(Vector<int>& v) {
   /* Base case: 0- or 1-element lists are
    * already sorted.
    */
   if (v.size() <= 1) {
      return;
   }

   /* 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);
}
```



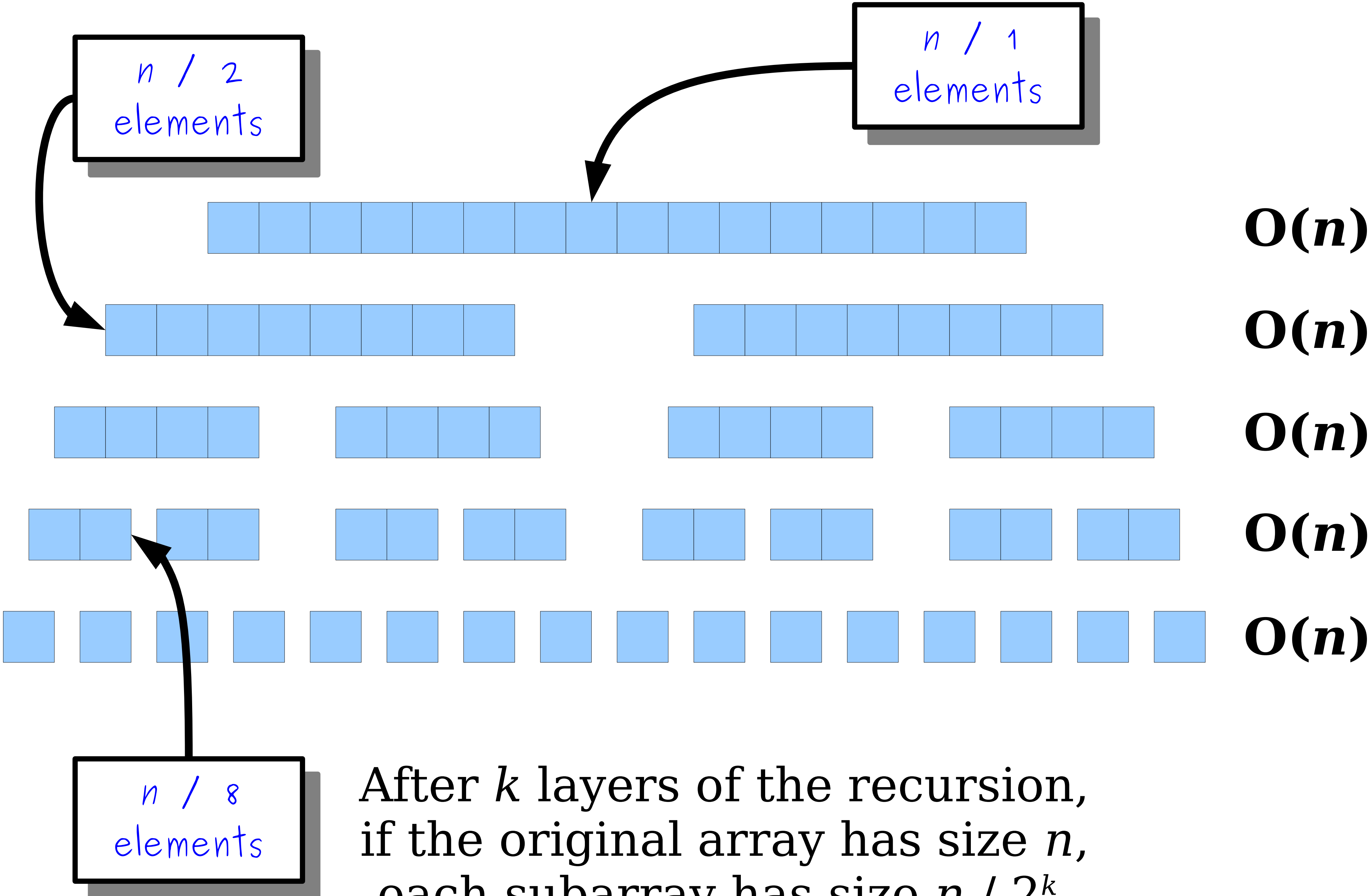
How much work does mergesort do at each level of recursion?



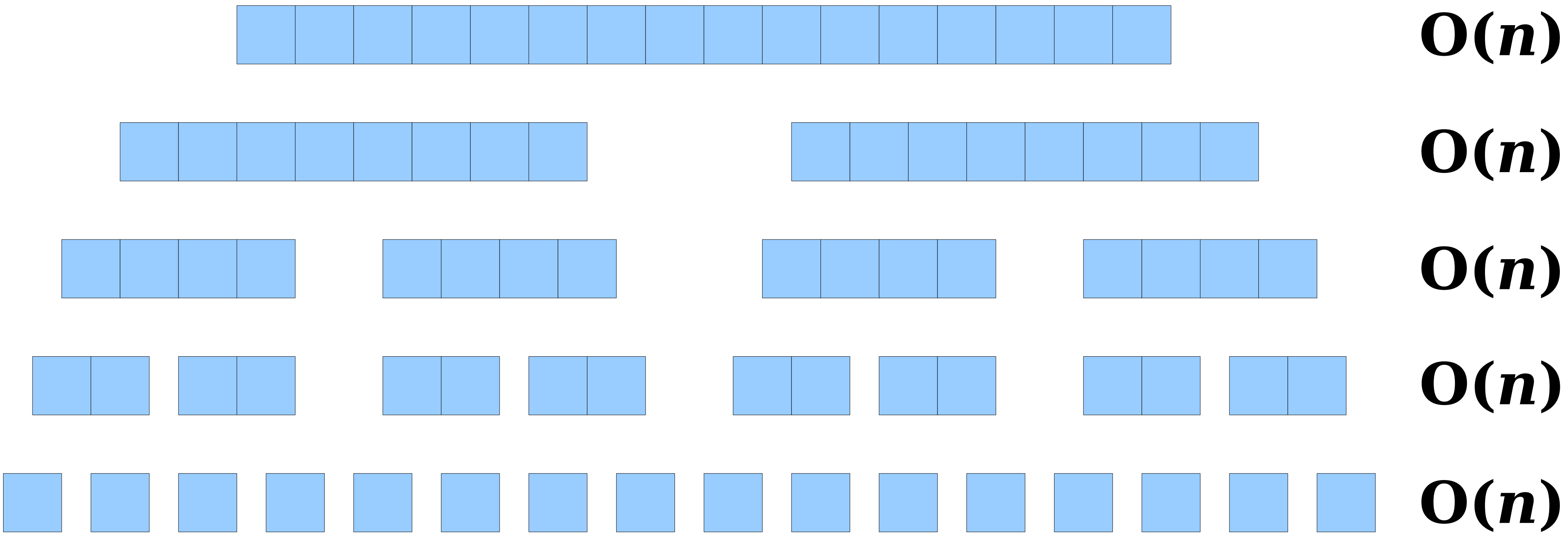
How many levels are there?



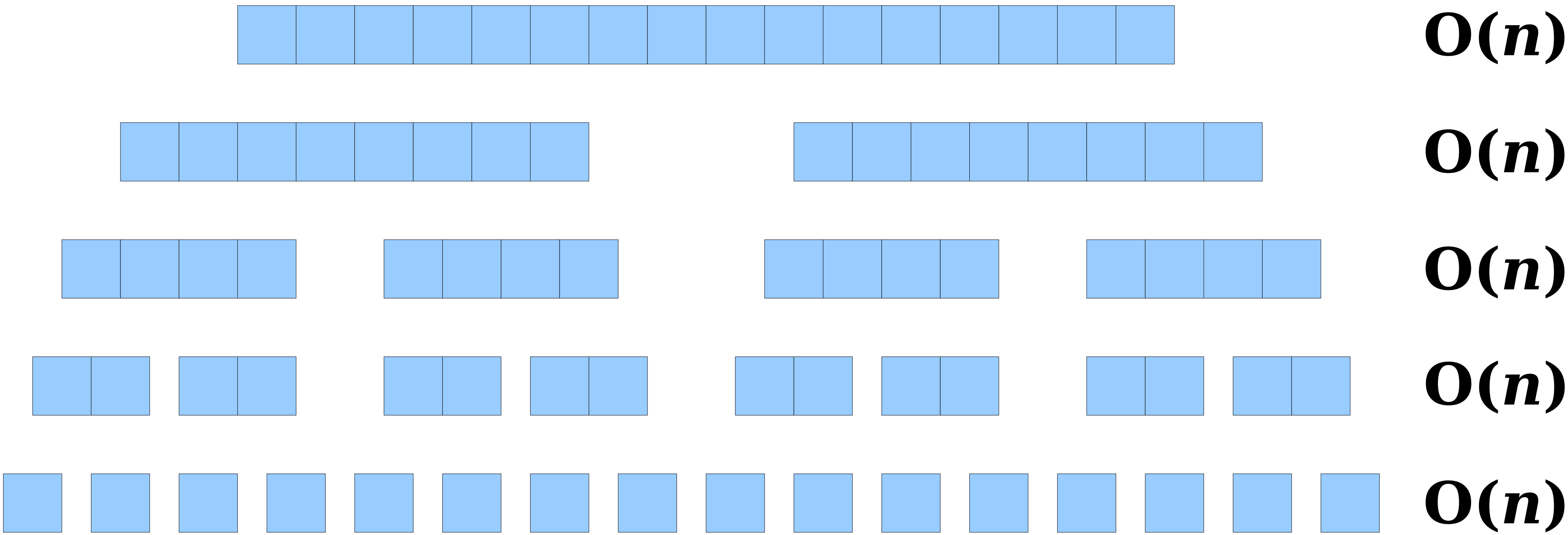
Each recursive call cuts the array size in half.



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



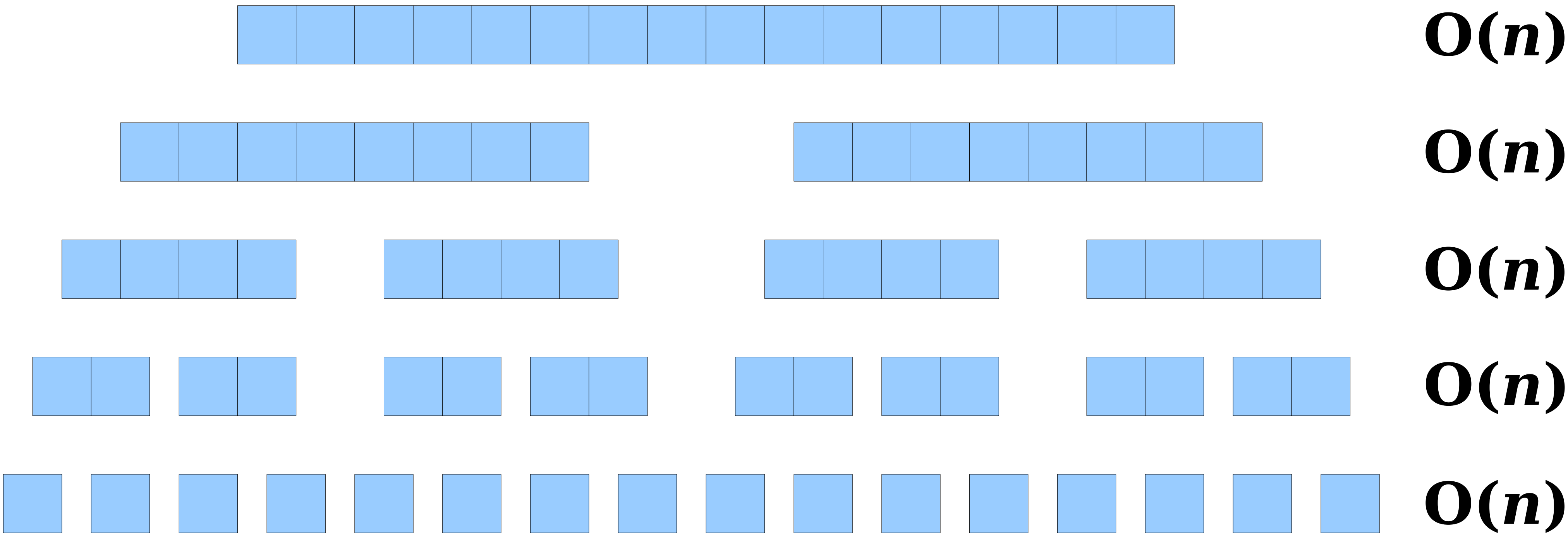
The recursion stops when we’re down to a single element.



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: $k = \log_2 n$.



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 Logistics

- The first midterm – which is a 48-hour take-home exam – goes out this Friday at 12:30PM and comes due this Sunday at 12:30PM (Pacific time).
- Information about the exam format, policies, etc. can be found at the online "Midterm Logistics" handout.
- ***We want you to do well on this exam***. We're impressed with how much progress you've made this quarter, and this is your chance to show off how much you've learned!

Practice Problems

- We have a collection of midterm practice problems available to help you prep.
- Check them out [on the course website].
- There are *way* more problems here than what we'd ask on a single exam; these problems are taken from five separate past midterm.

WiCS Study Night

- Stanford Women in Computer Science (WiCS) is holding a study night for CS106B tonight (Wednesday, February 10th) at 5PM PST.
- This event is open to everyone – feel free to join if you're interested!
- Call in using Nooks via this link.

Getting Help

- As you're studying, ***please keep us in the loop!***
- We're happy to review code you've written for the practice problems, to go over concepts we talked about in past lectures, etc.

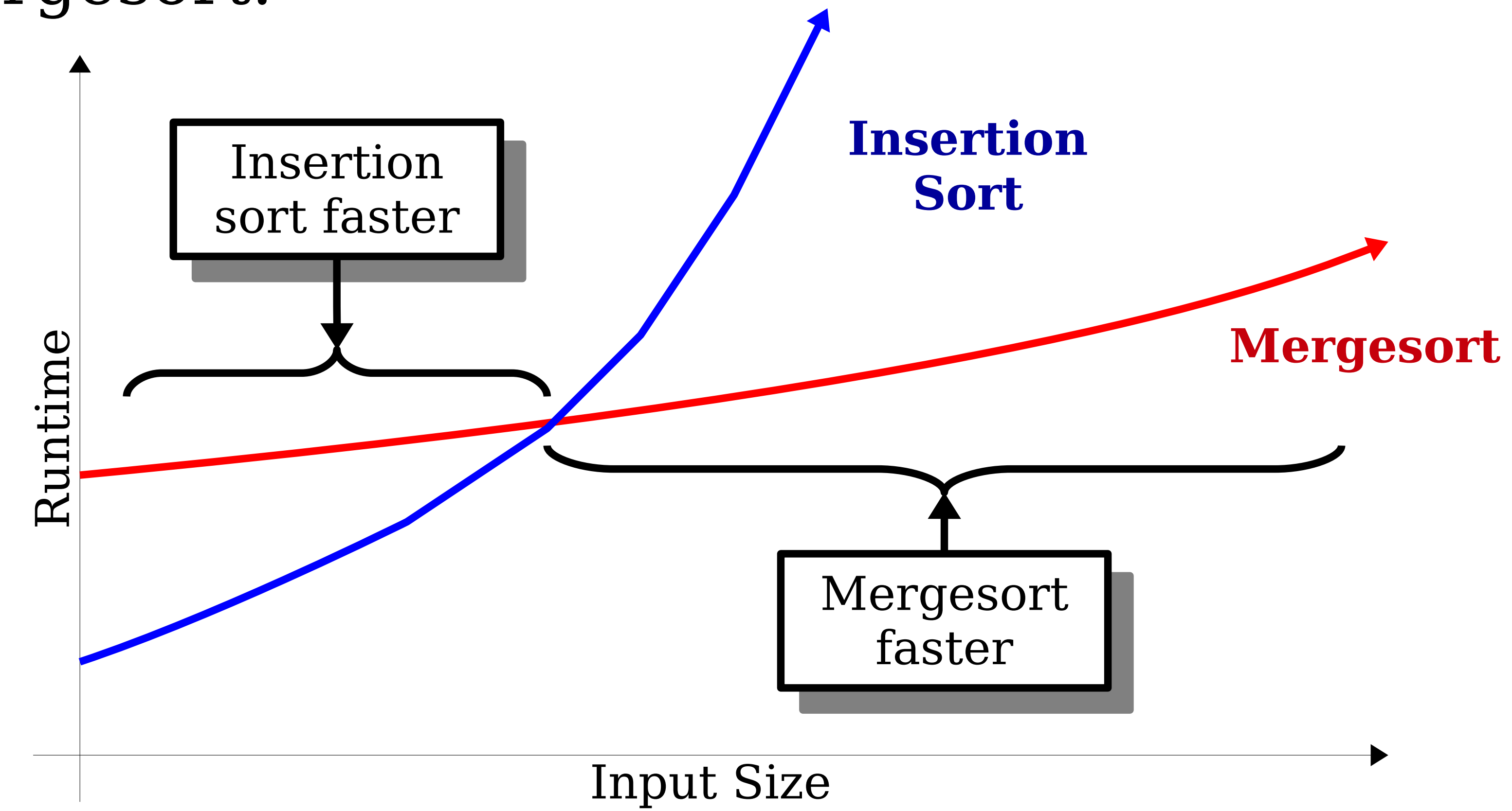
```
lecture.notify_all();
```

(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?

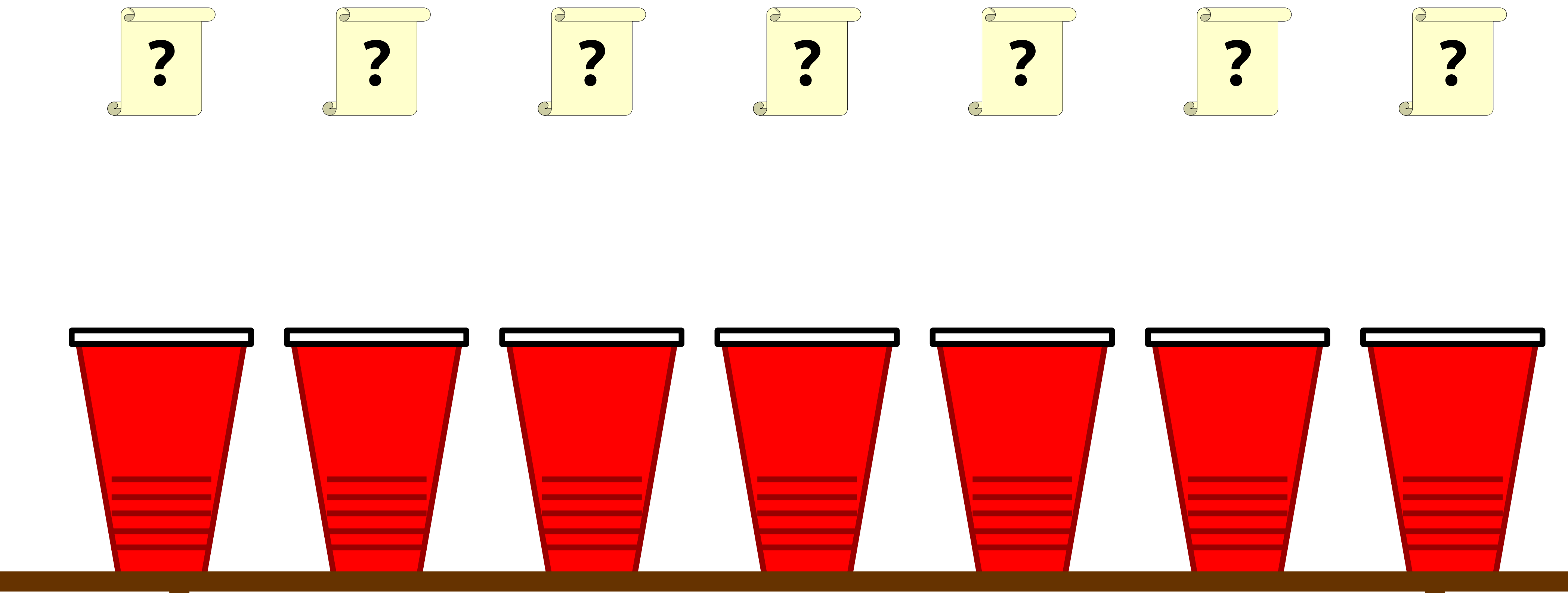
Suppose we want to search an array for an element, and we know that array is sorted.

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

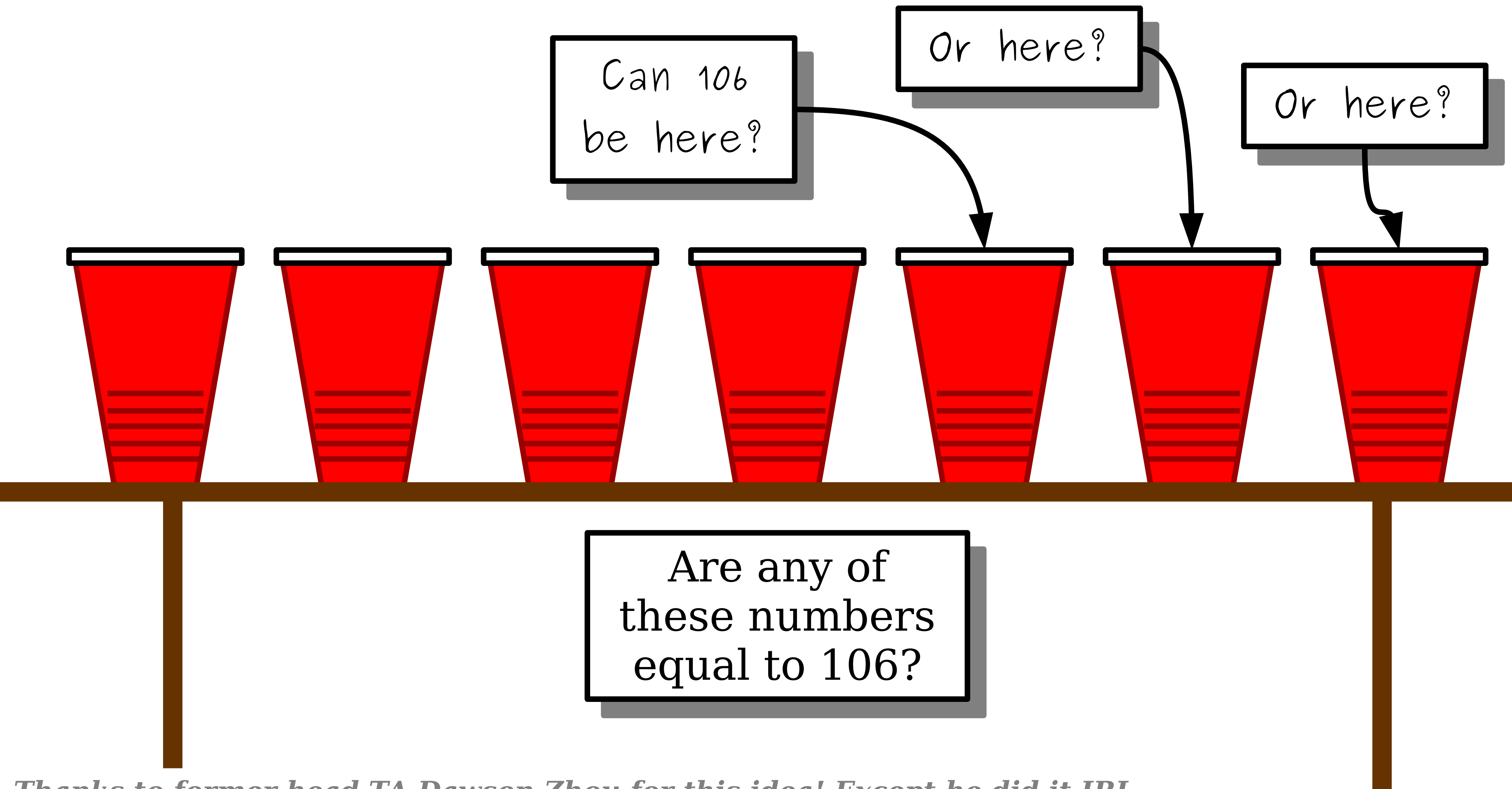


Are any of these numbers equal to 106?

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

Each cup contains a number.

Numbers are sorted from left to right

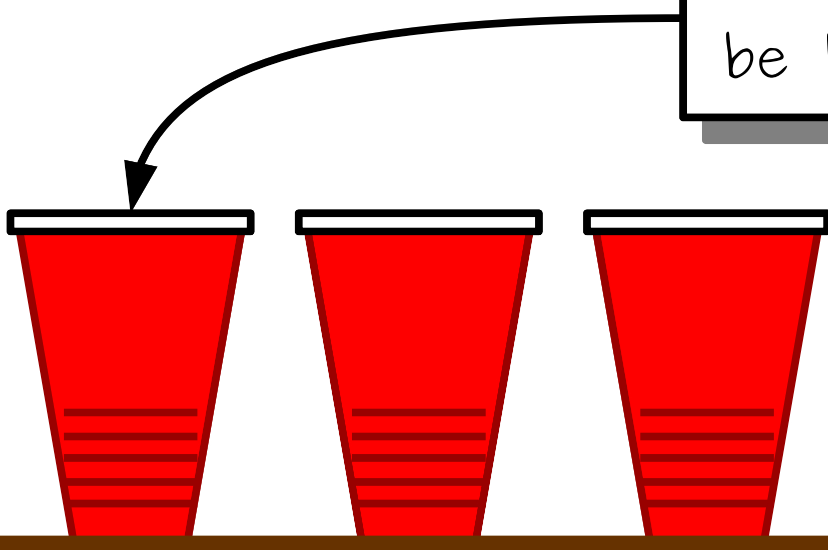


Thanks to former head TA Dawson Zhou for this idea! Except he did it IRL.

Each cup contains a number.

Numbers are sorted from left to right

Can 106 be here?

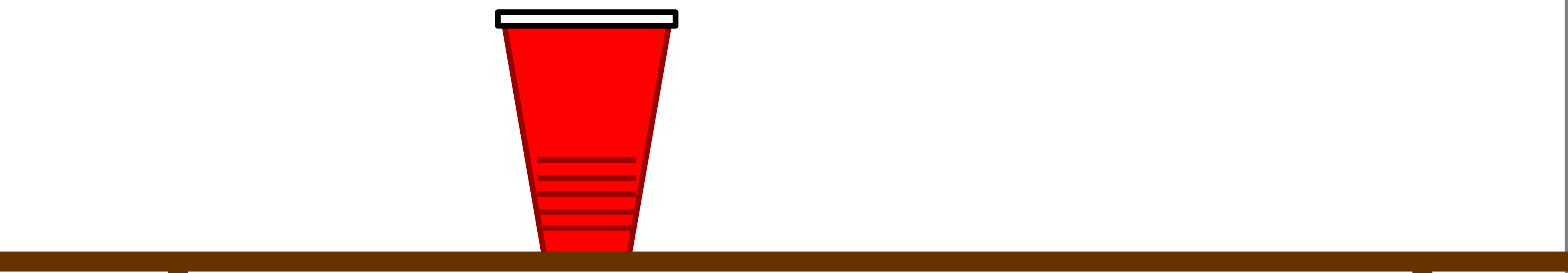


Are any of these numbers equal to 106?

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

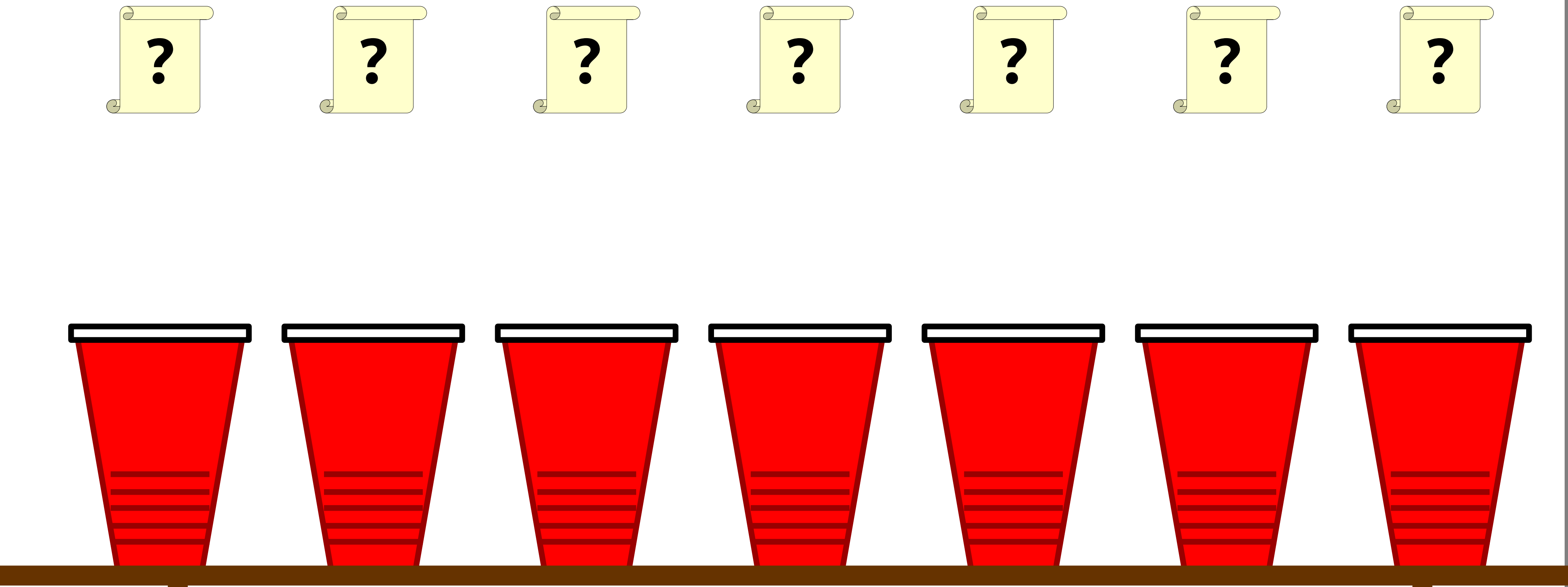
Alas, 106 is not to be found here.

Are any of these numbers equal to 106?

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

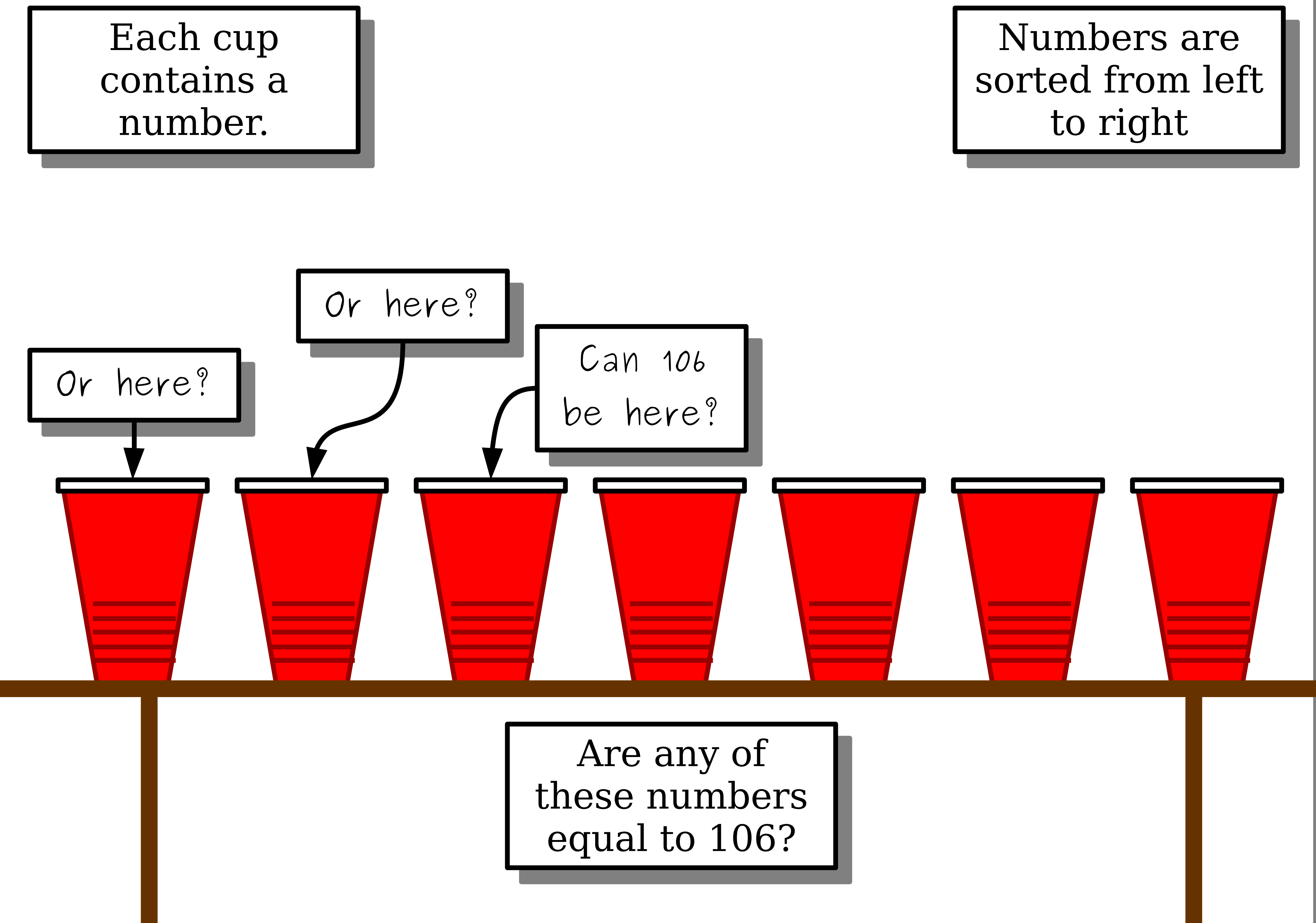
Each cup contains a number.

Numbers are sorted from left to right



Are any of these numbers equal to 106?

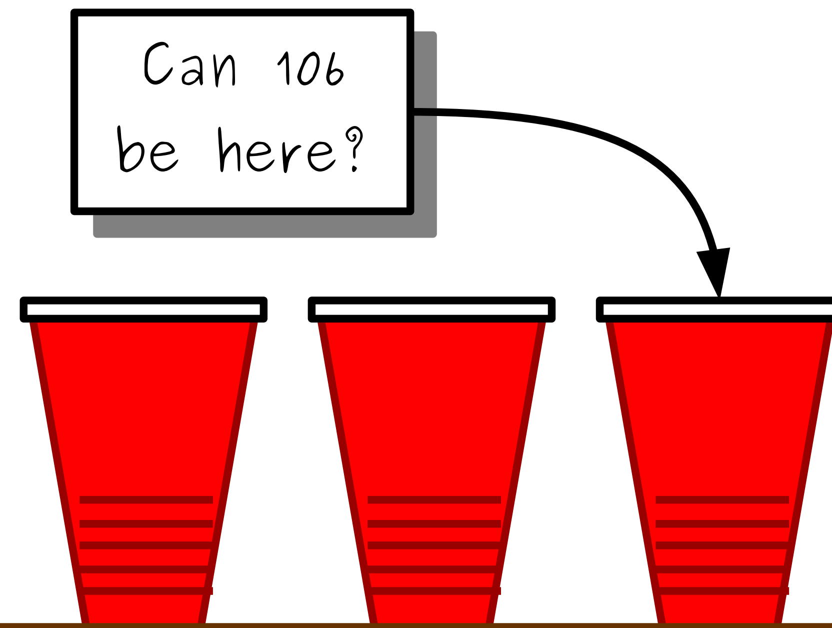
Thanks to former head TA Dawson Zhou for this idea! Except he did it IRL.



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



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.***
  - There’s a bunch of goodies about big-O, searching, and sorting in there we didn’t have time to explore here.
- ***Study for the Midterm.***
  - There’s a practice exam available online. You’ve got the section handouts. And you can come talk to us!

Next Time

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