# Searching and Sorting

## Part One

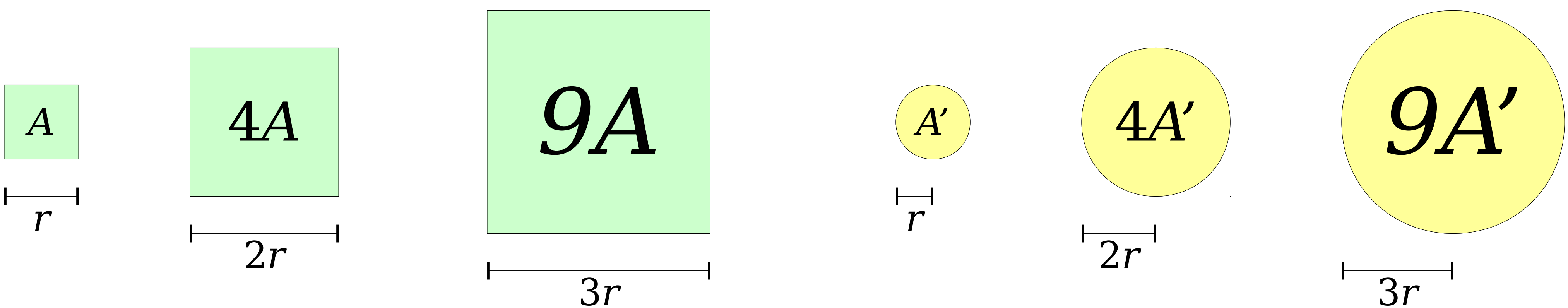
# *Apply to Section Lead!*

***Don’t just take my word for it…***

# Recap from Last Time

# Big-O Notation

- ***Big-O notation*** is a way of quantifying the rate at which some quantity grows.
- For example:
  - A square of side length $r$ has area $O(r^2)$.
  - A circle of radius $r$ has area $O(r^2)$.




Doubling $r$ increases area 4×.
Tripling $r$ increases area 9×.

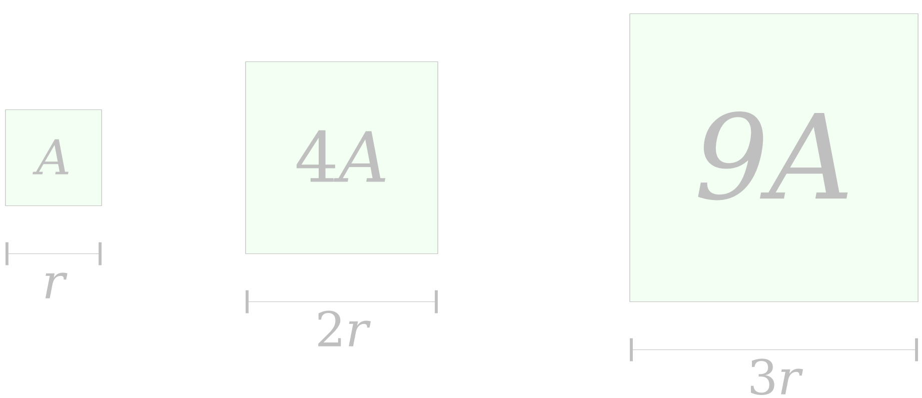
Doubling $r$ increases area 4×.
Tripling $r$ increases area 9×.

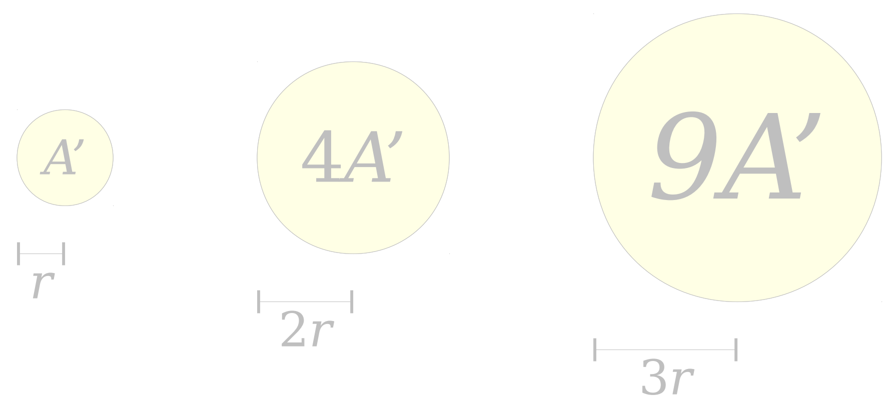
# Big-O Notation

- ***Big-O notation*** is a w
  rate at which some qu

This just says that these quantities grow at the same relative rates. It does not say that they're equal!

- For example:
  - A square of side length $r$ has area $O(r^2)$.
  - A circle of radius $r$ has area $O(r^2)$.




Doubling $r$ increases area 4×.
Tripling $r$ increases area 9×.

Doubling $r$ increases area 4×.
Tripling $r$ increases area 9×.

```
double averageOf(const Vector<int>& vec) {
    double total = 0.0;

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

    return total / vec.size();
}
```

---

Assume any individual statement takes one unit of time to execute. If the input `Vector` has $n$ elements, how many time units will this code take to run?

```
    double averageOf(const Vector<int>& vec) {
  1   double total = 0.0;

                1              n+1          n
      for (int i = 0; i < vec.size(); i++) {
          total += vec[i];
      }         n


      return total / vec.size(); 1
    }
```

Assume any individual statement takes one unit of time to execute. If the input `Vector` has $n$ elements, how many time units will this code take to run?

```
double averageOf(const Vector<int>& vec) {
1 double total = 0.0;

                1          n+1        n
  for (int i = 0; i < vec.size(); i++) {
      total += vec[i];
  }               n

  return total / vec.size(); 1
}
```

One possible answer: $3n + 4$.

```
double averageOf(const Vector<int>& vec) {
1 double total = 0.0;

             1             n+1        n
  for (int i = 0; i < vec.size(); i++) {
      total += vec[i];
  }          n

  return total / vec.size(); 1
}
```

~~One possible answer: $3n + 4$.~~

More useful answer: **$O(n)$**.

```
void printStars(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << '*' << endl;
        }
    }
}
```

Work Done: **O($n^2$)**.

```
void hmmThatsStrange(int n) {
    cout << "Mirth and Whimsy" << endl;
}
```

Work Done: **O(1)**.

# New Stuff!

# Sorting Algorithms

What is sorting?


One style of
“sorting,” but not
the one we’re
thinking about…

| Time | Auto | Athlete | Nationality | Date | Venue |
|---|---|---|---|---|---|
| **4:37.0** | | Anne Smith | United Kingdom | 3 June 1967[8] | London |
| **4:36.8** | | Maria Gommers | Netherlands | 14 June 1969[8] | Leicester |
| **4:35.3** | | Ellen Tittel | West Germany | 20 August 1971[8] | Sittard |
| **4:29.5** | | Paola Pigni | Italy | 8 August 1973[8] | Viareggio |
| **4:23.8** | | Natalia Mărășescu | Romania | 21 May 1977[8] | Bucharest |
| **4:22.1** | 4:22.09 | Natalia Mărășescu | Romania | 27 January 1979[8] | Auckland |
| **4:21.7** | 4:21.68 | Mary Decker | United States | 26 January 1980[8] | Auckland |
| | **4:20.89** | Lyudmila Veselkova | Soviet Union | 12 September 1981[8] | Bologna |
| | **4:18.08** | Mary Decker-Tabb | United States | 9 July 1982[8] | Paris |
| | **4:17.44** | Maricica Puică | Romania | 9 September 1982[8] | Rieti |
| | **4:16.71** | Mary Decker-Slaney | United States | 21 August 1985[8] | Zürich |
| | **4:15.61** | Paula Ivan | Romania | 10 July 1989[8] | Nice |
| | **4:12.56** | Svetlana Masterkova | Russia | 14 August 1996[8] | Zürich |
| | **4:12.33** | Sifan Hassan | Netherlands | 12 July 2019 | Monaco |

***Problem:*** Given a list of data points, sort those data points into ascending / descending order by some quantity.

Suppose we want to rearrange a sequence to put elements into ascending order.

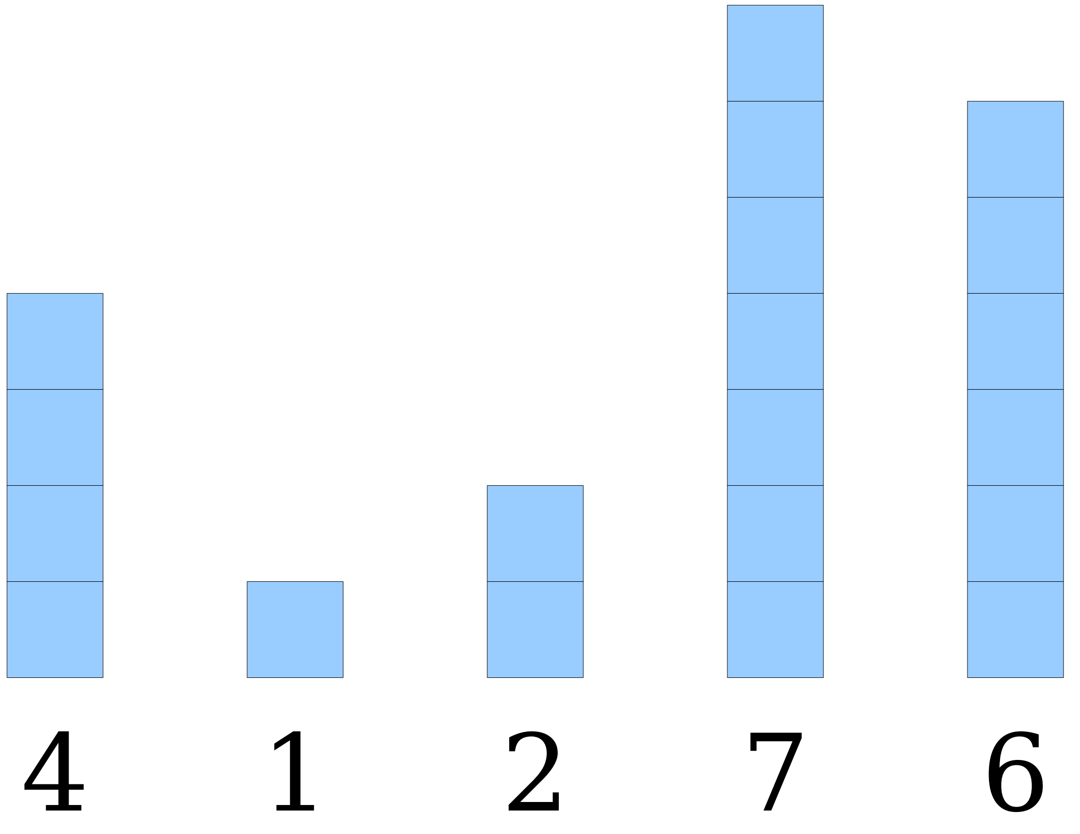
What are some strategies we could use?

How do those strategies compare?

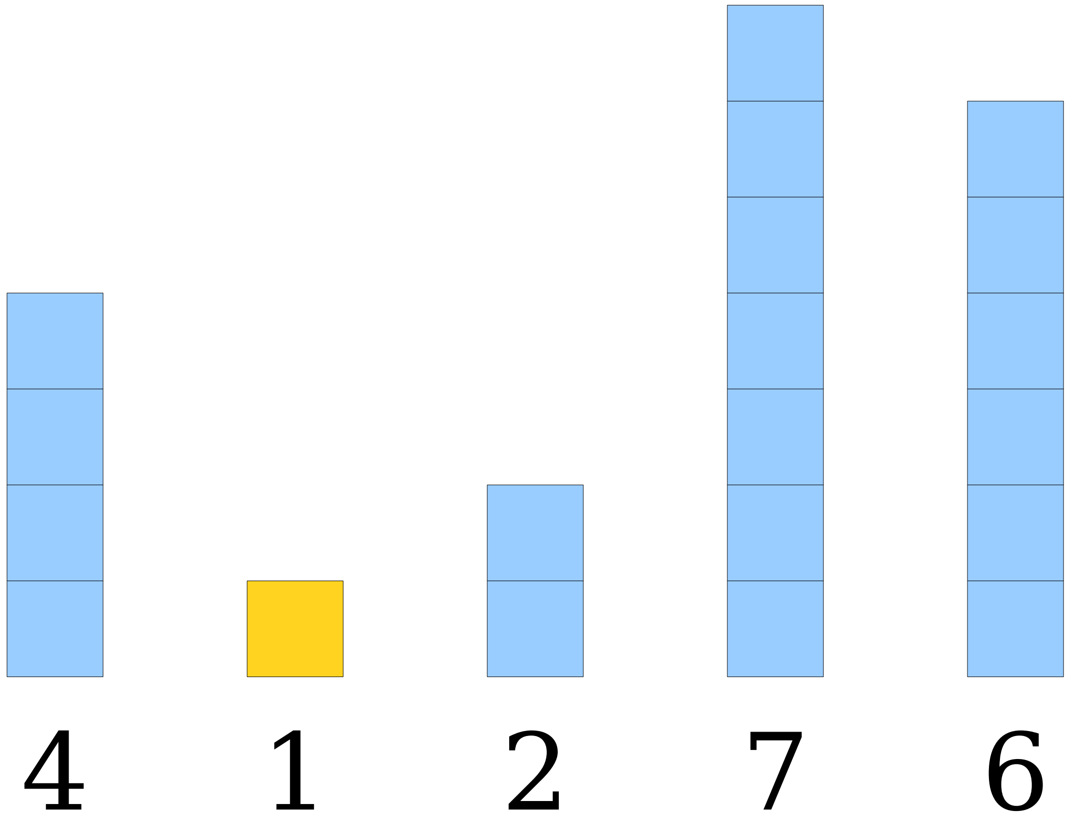
Is there a "best" strategy?

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

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

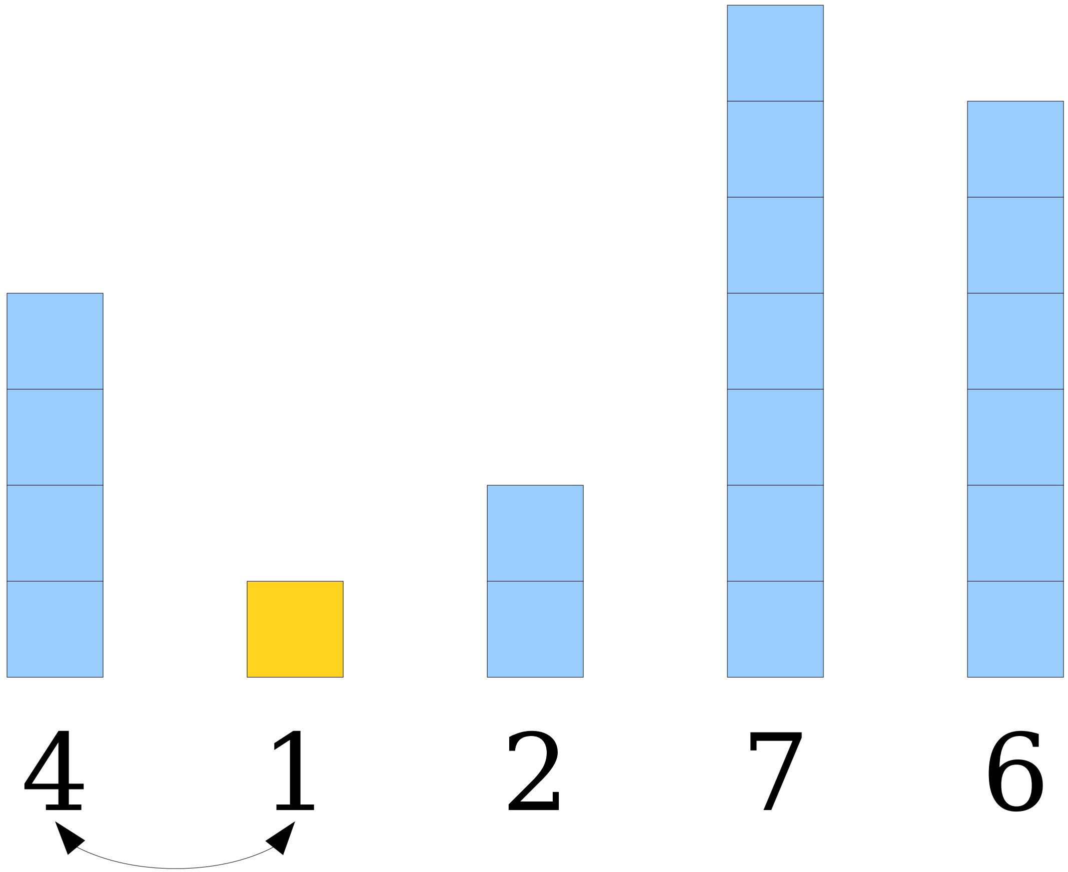


# An Initial Idea: ***Selection Sort***

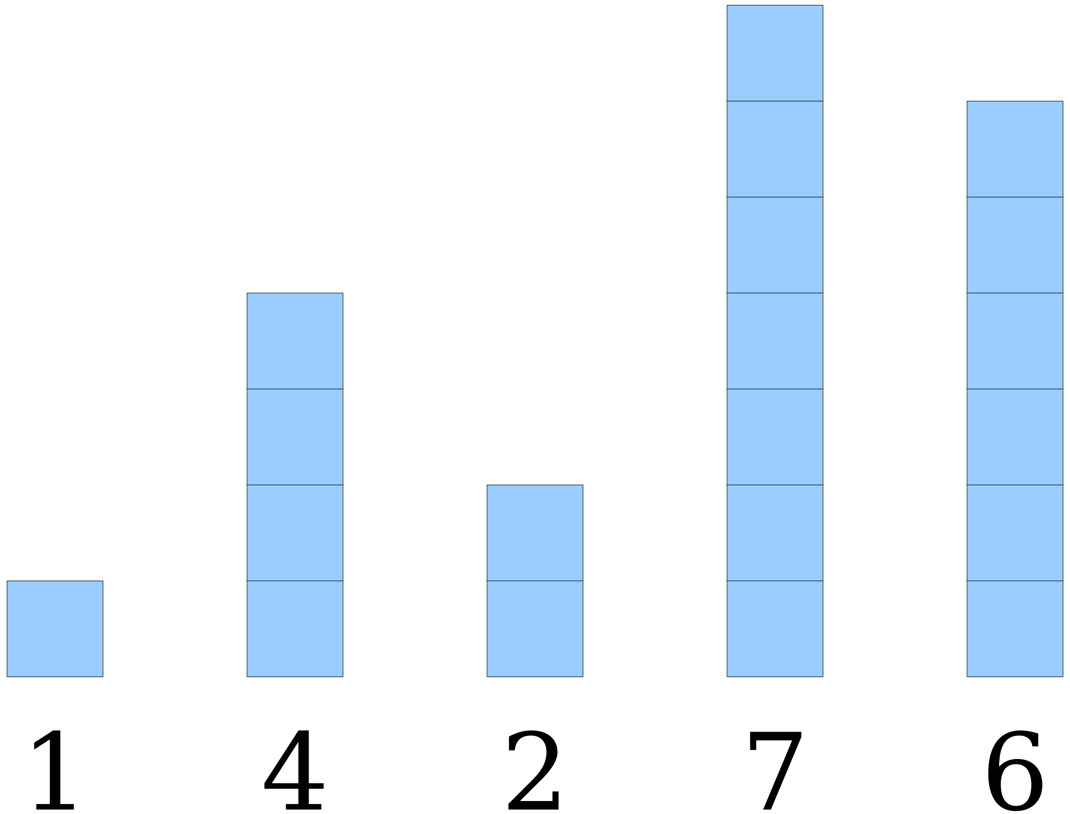



The smallest element should go in front.

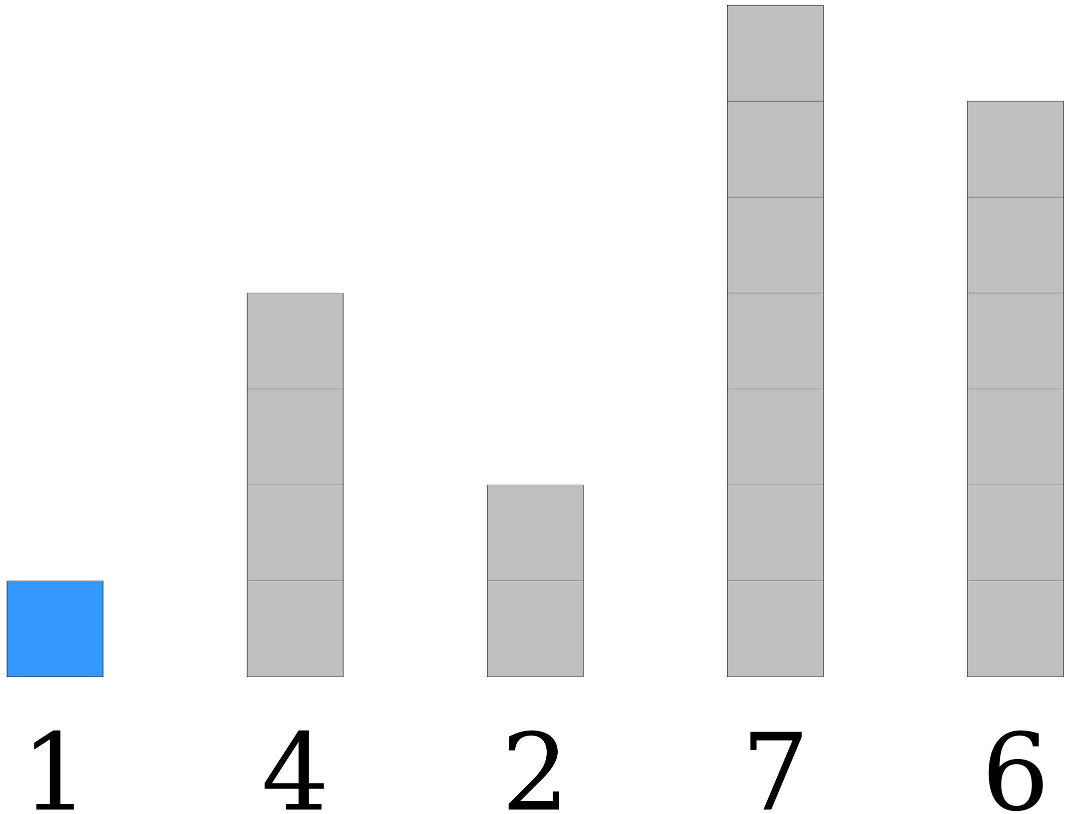
# An Initial Idea: ***Selection Sort***



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



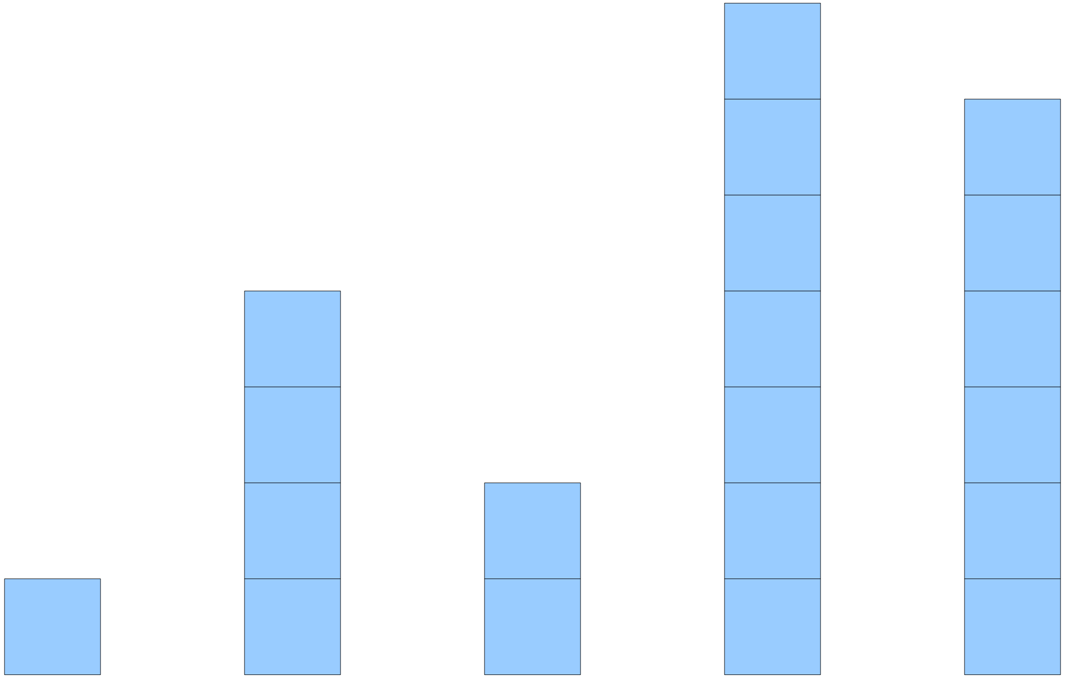
# An Initial Idea: ***Selection Sort***




This element is in the right place now.

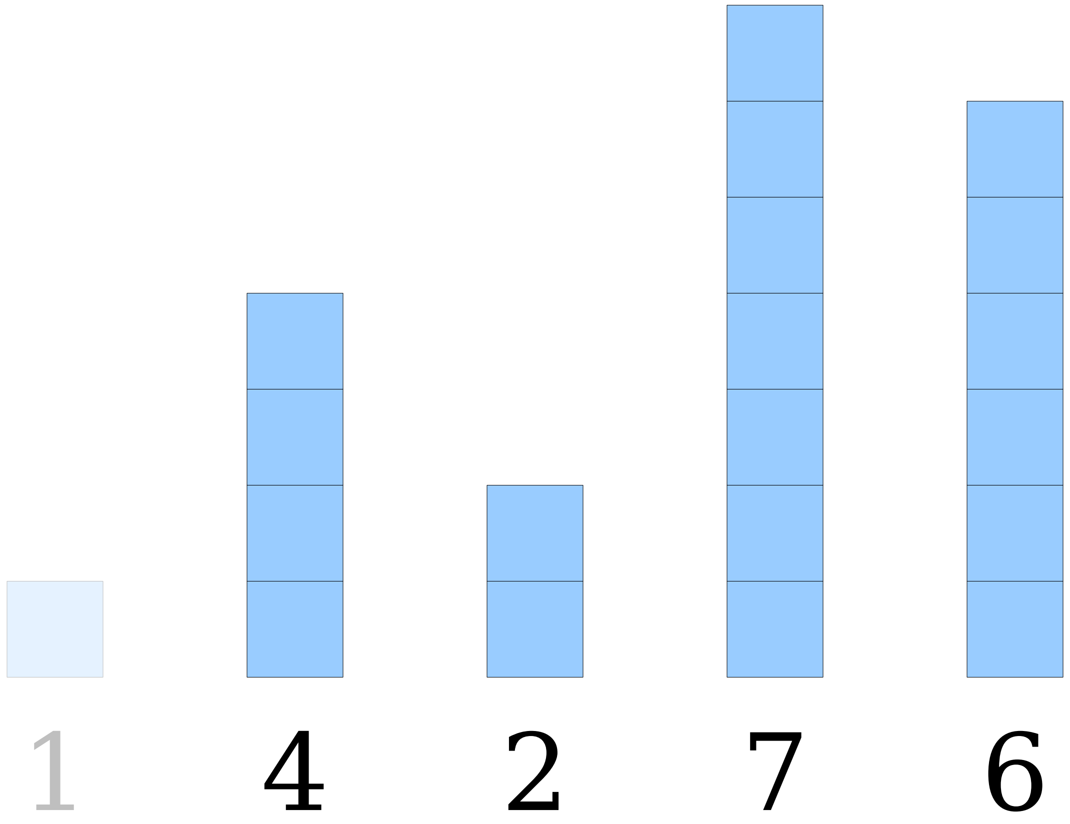
The remaining elements are in no particular order.

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

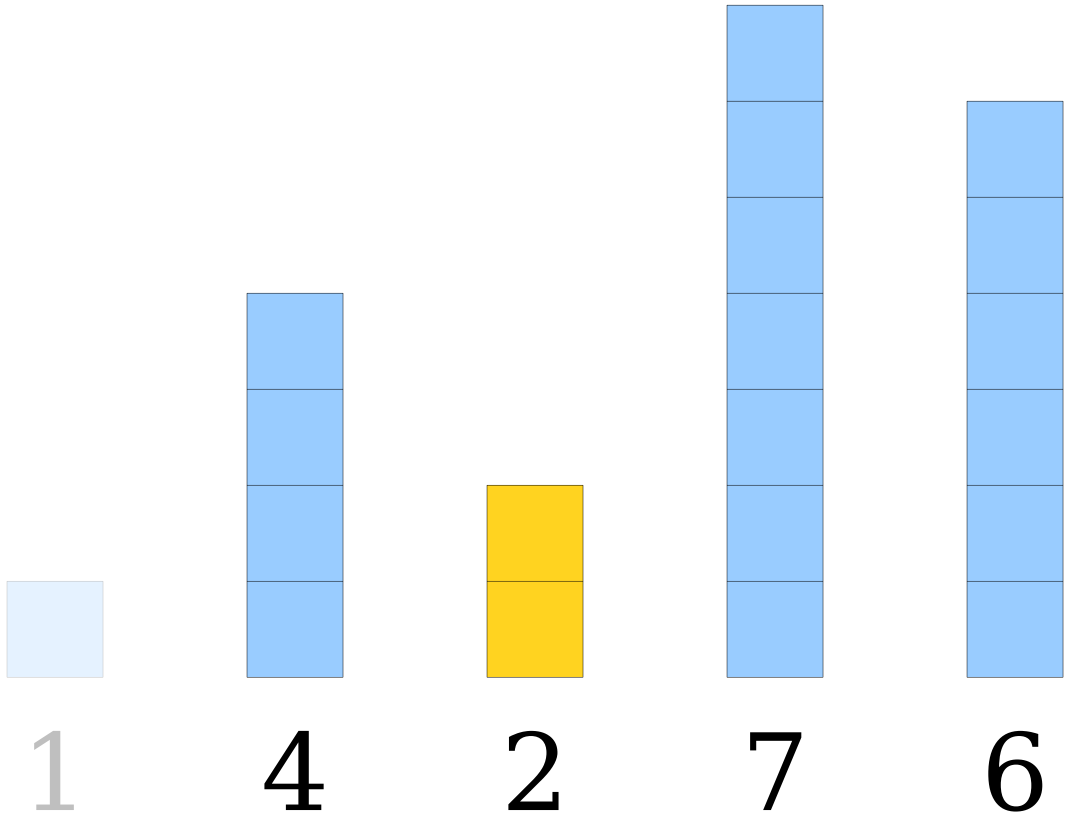


1 4 2 7 6

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

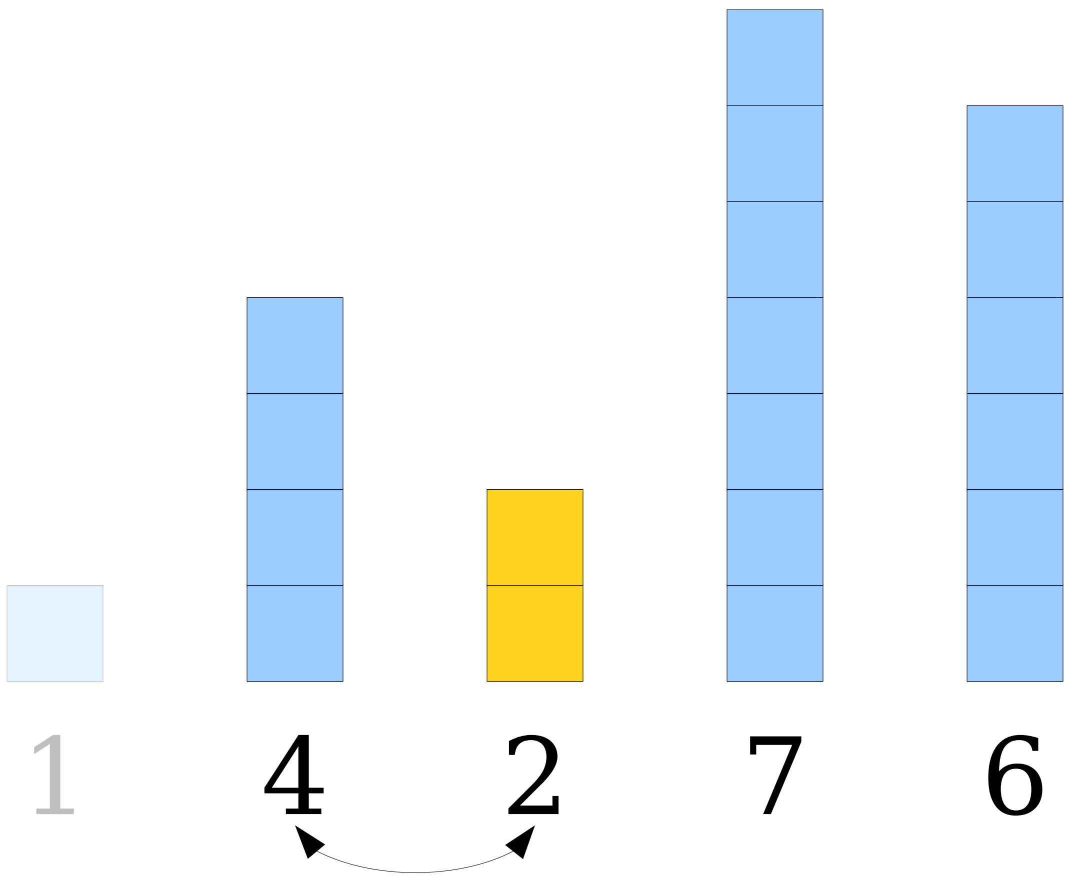


# An Initial Idea: ***Selection Sort***

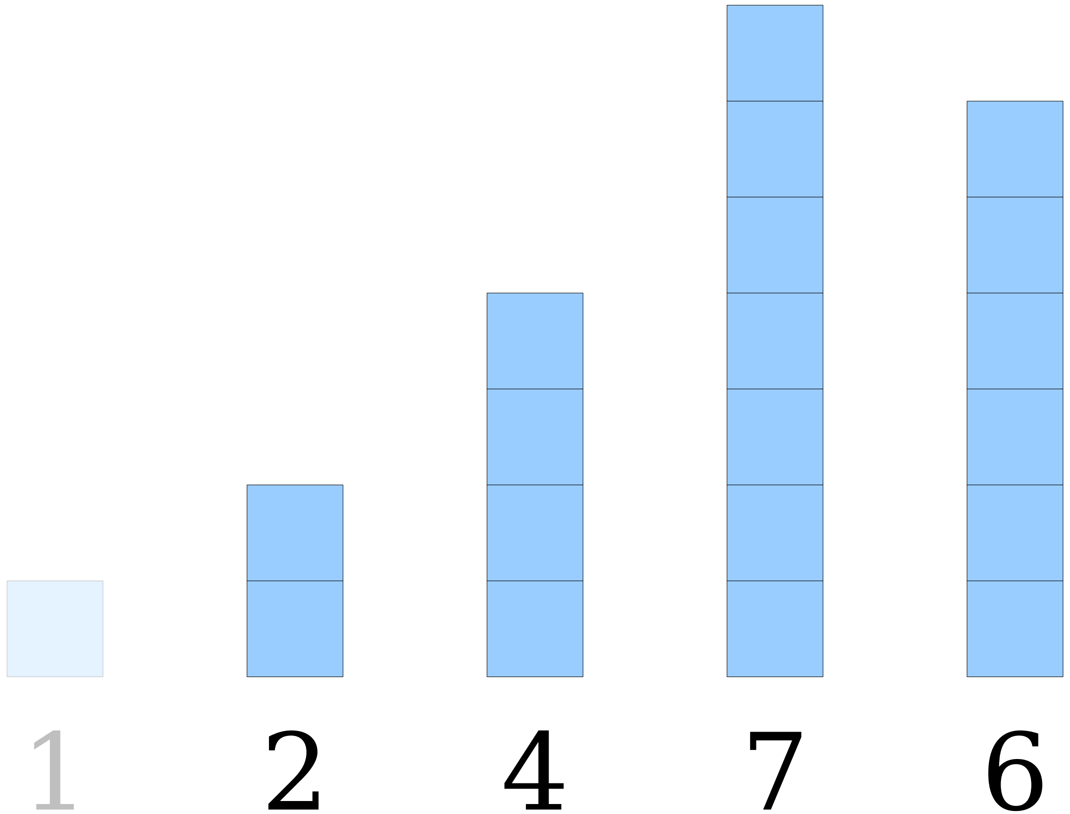



The smallest element of the remaining elements goes at the front of the remaining elements.

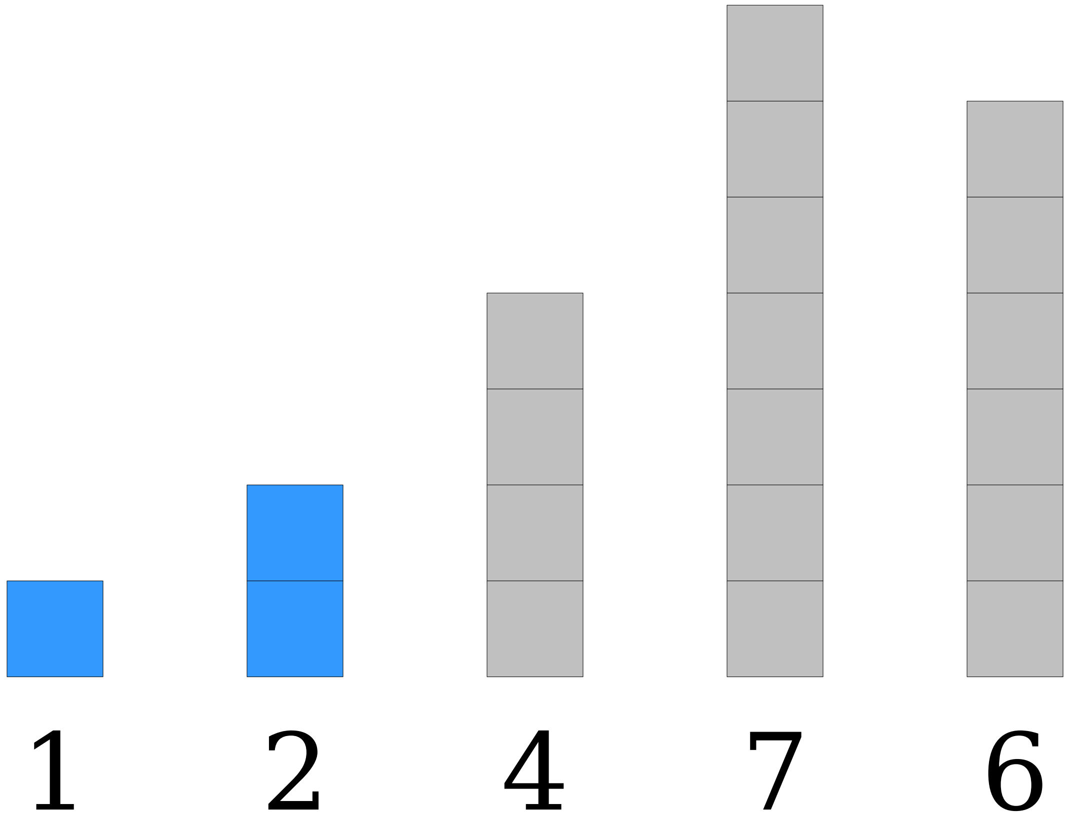
# An Initial Idea: ***Selection Sort***



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



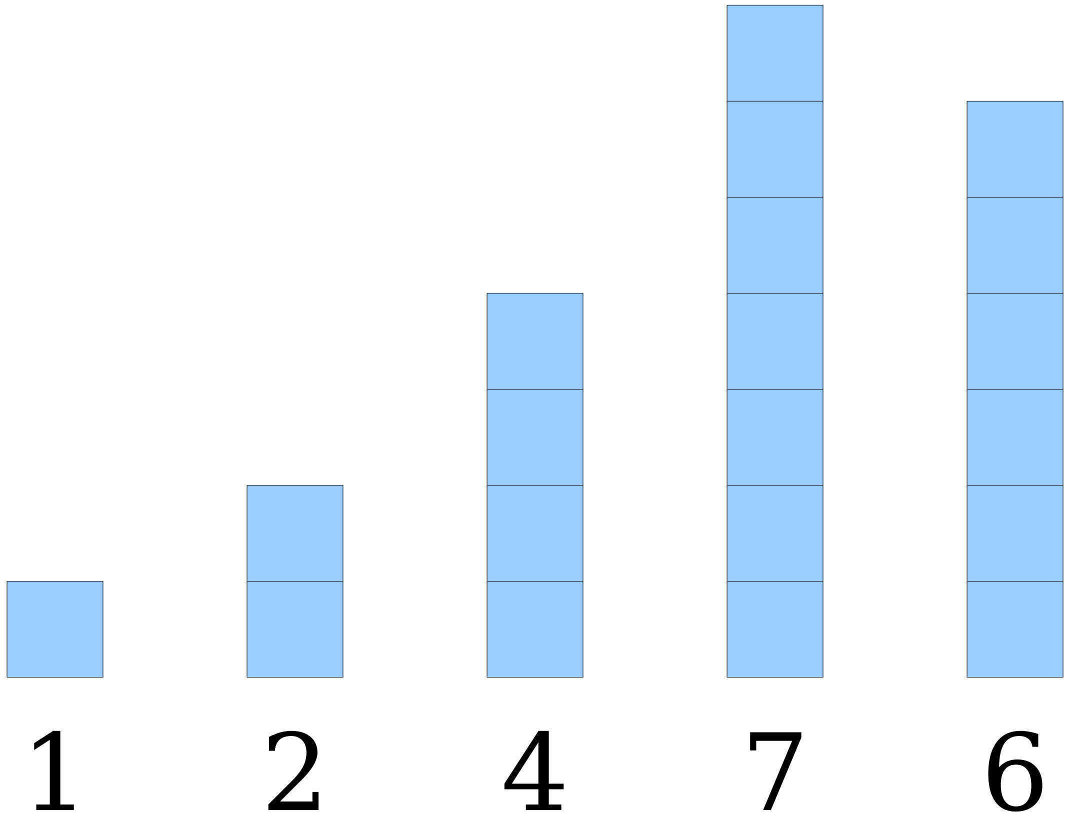
# An Initial Idea: ***Selection Sort***



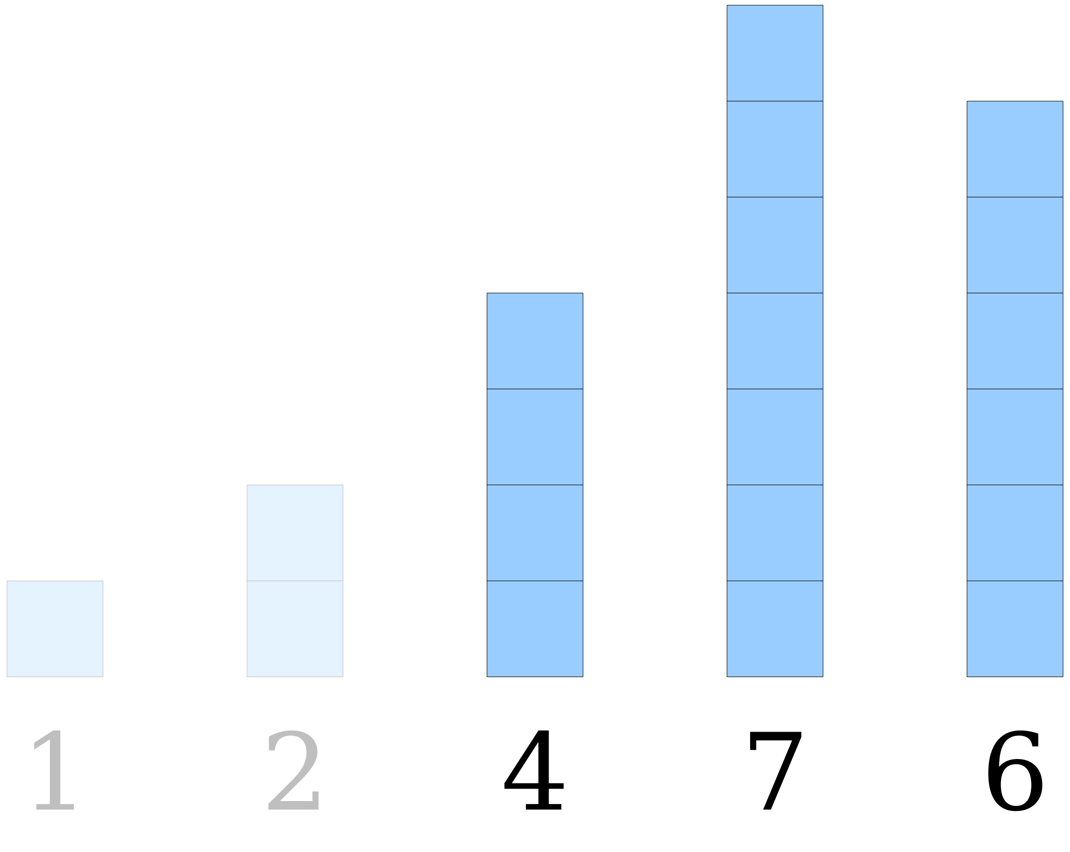

These elements are in the right place now.

The remaining elements are in no particular order.

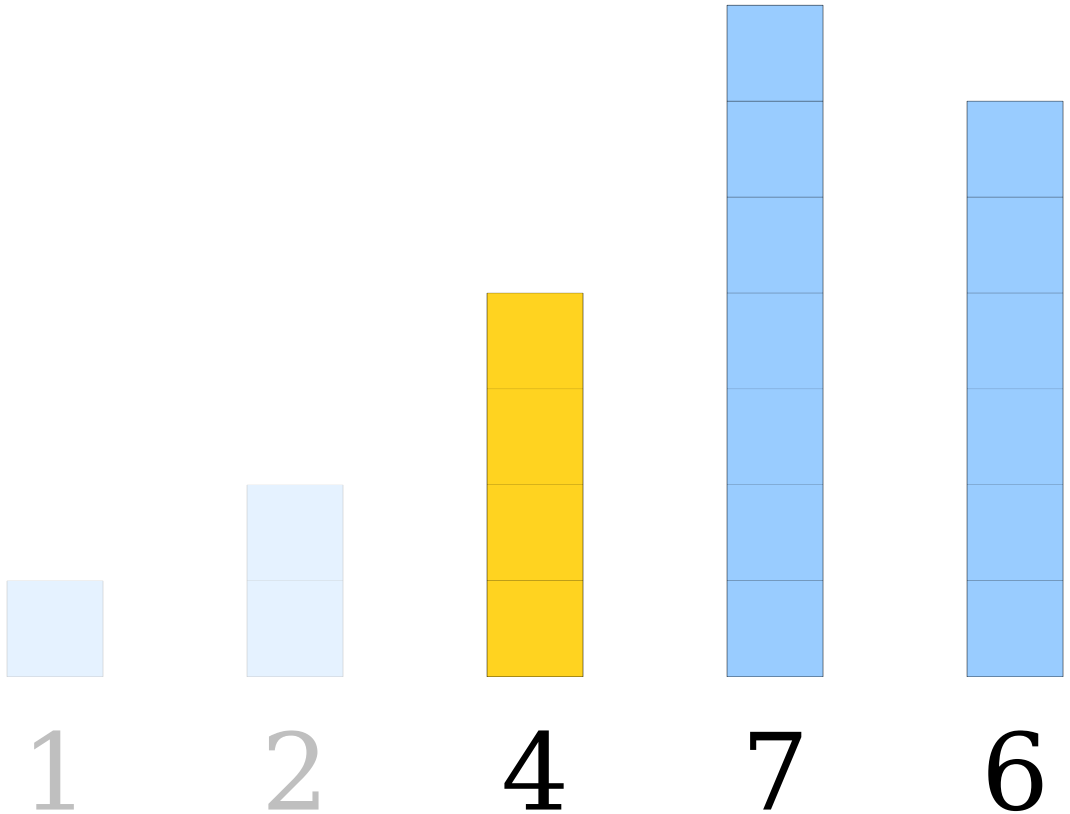
# An Initial Idea: ***Selection Sort***



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

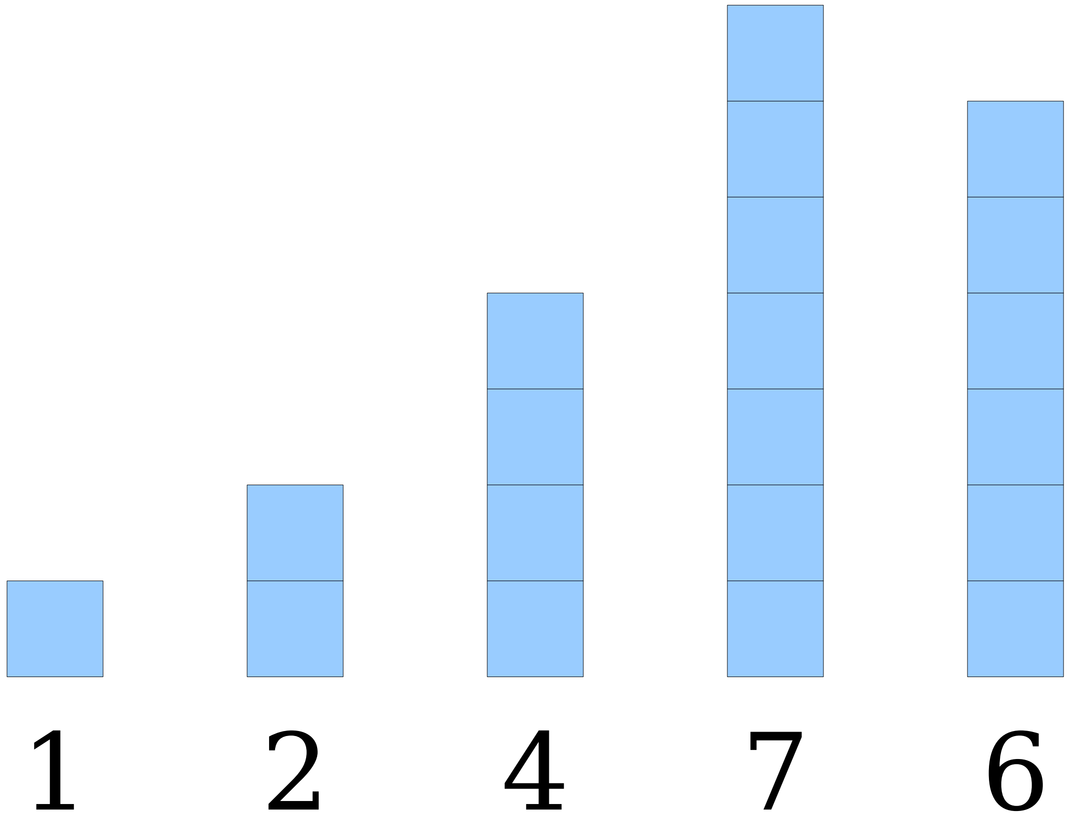


# An Initial Idea: ***Selection Sort***

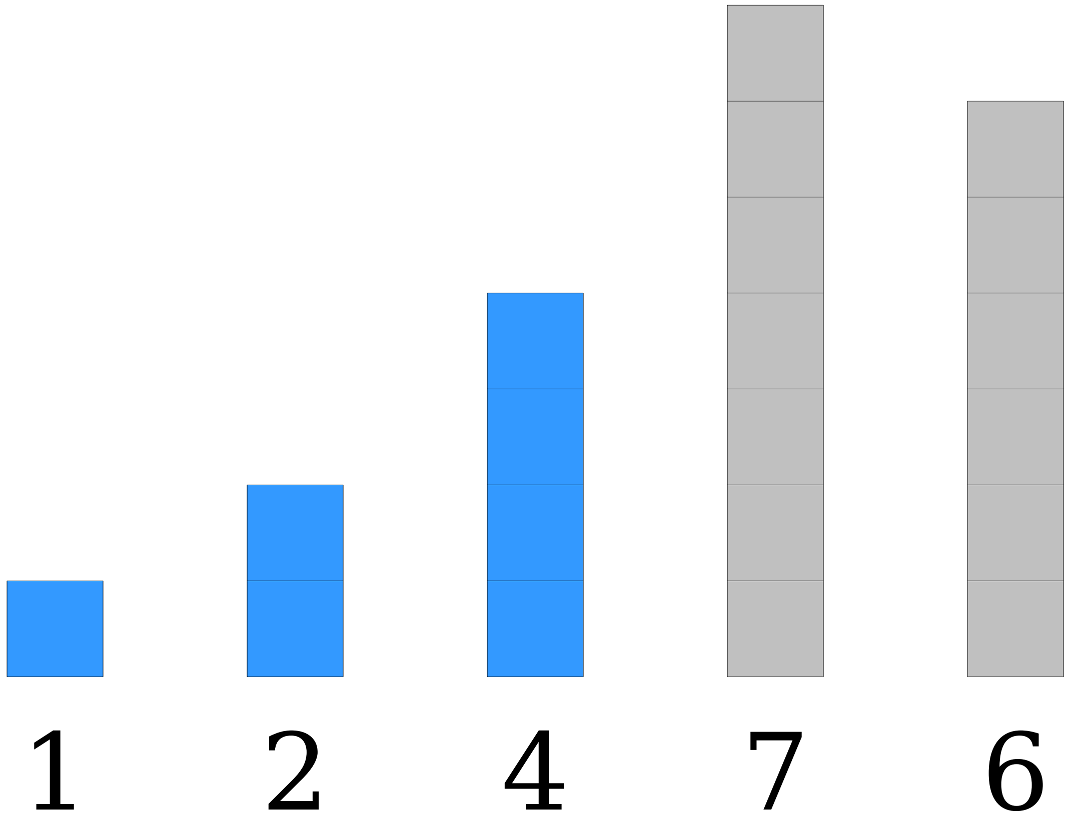



The smallest of these remaining elements goes at the front of the remaining elements.

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



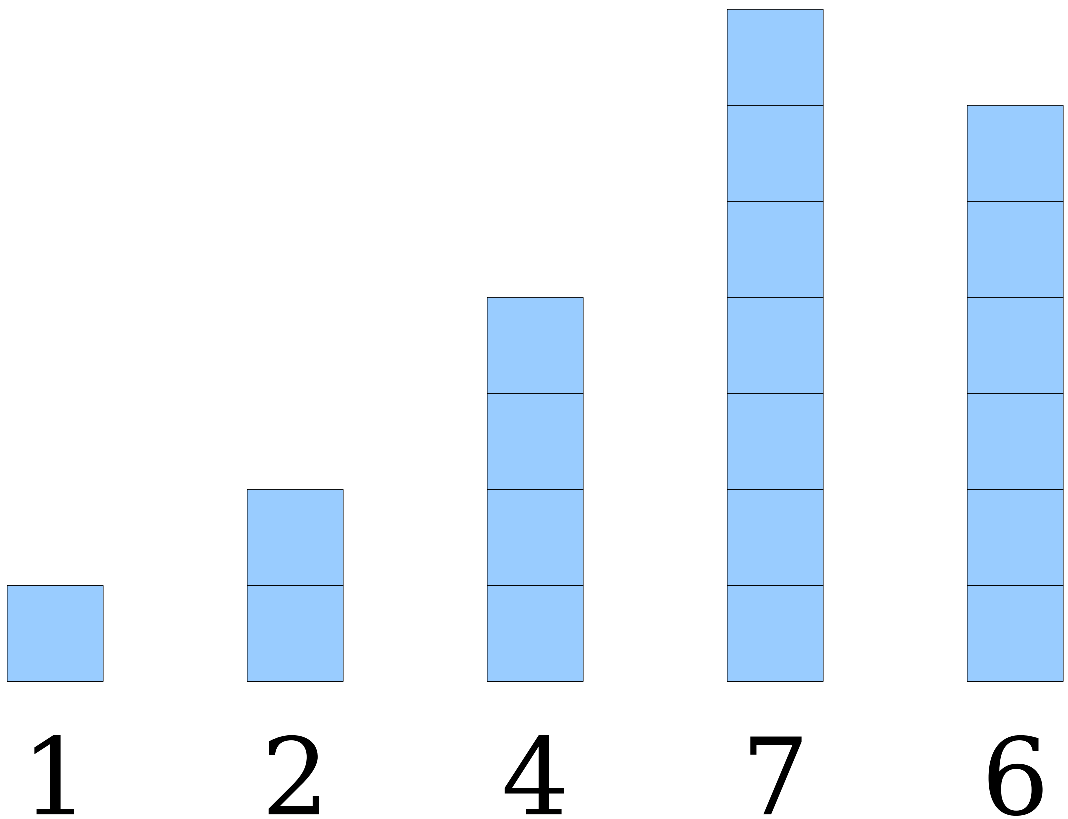
# An Initial Idea: ***Selection Sort***



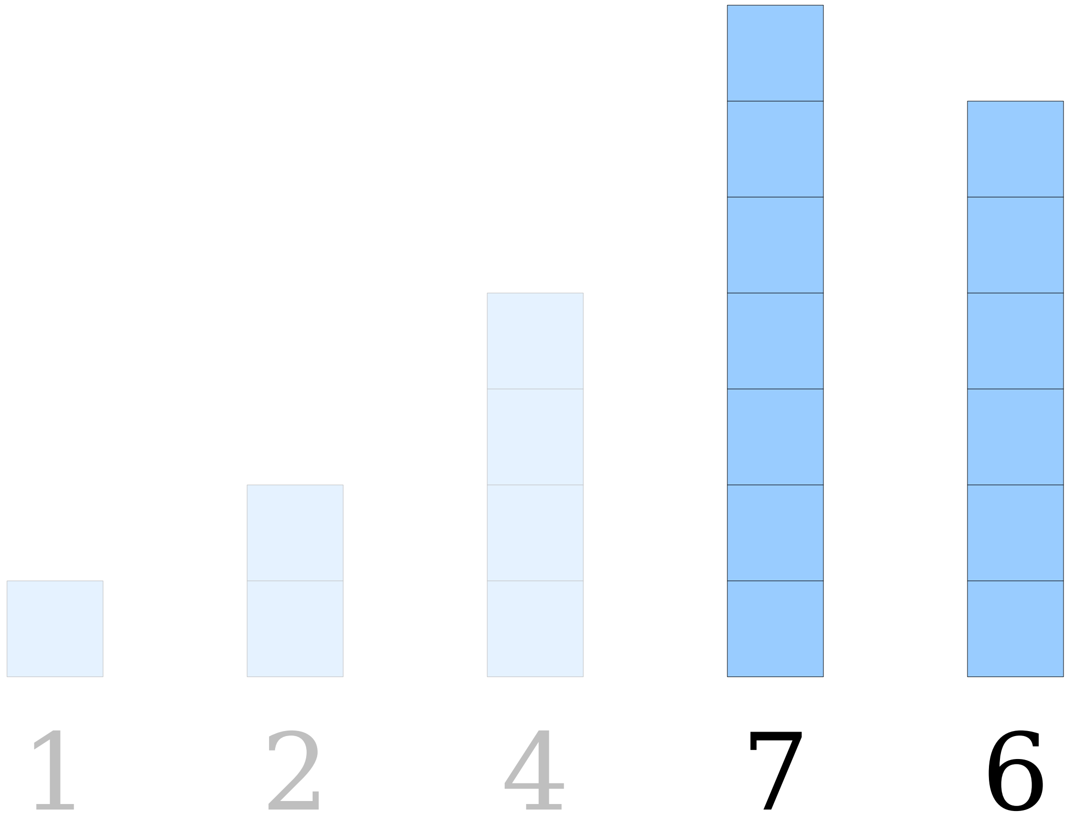

These elements are in the right place now.

The remaining elements are in no particular order.

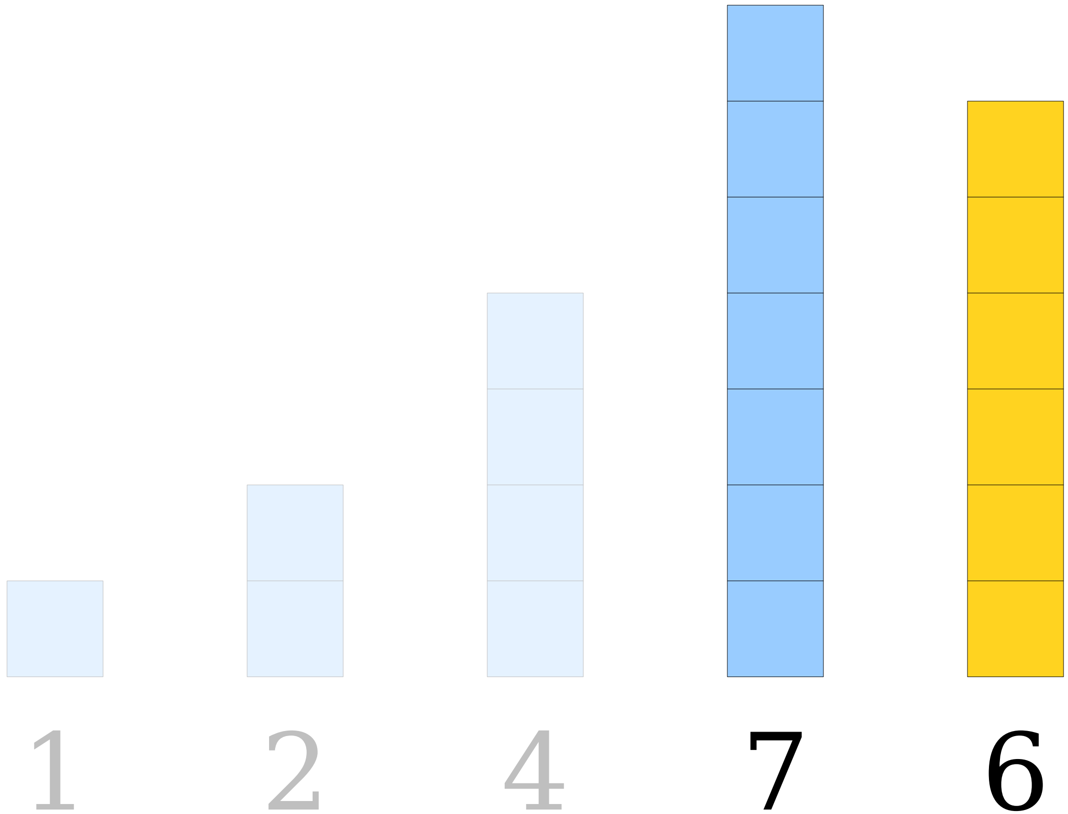
# An Initial Idea: ***Selection Sort***



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

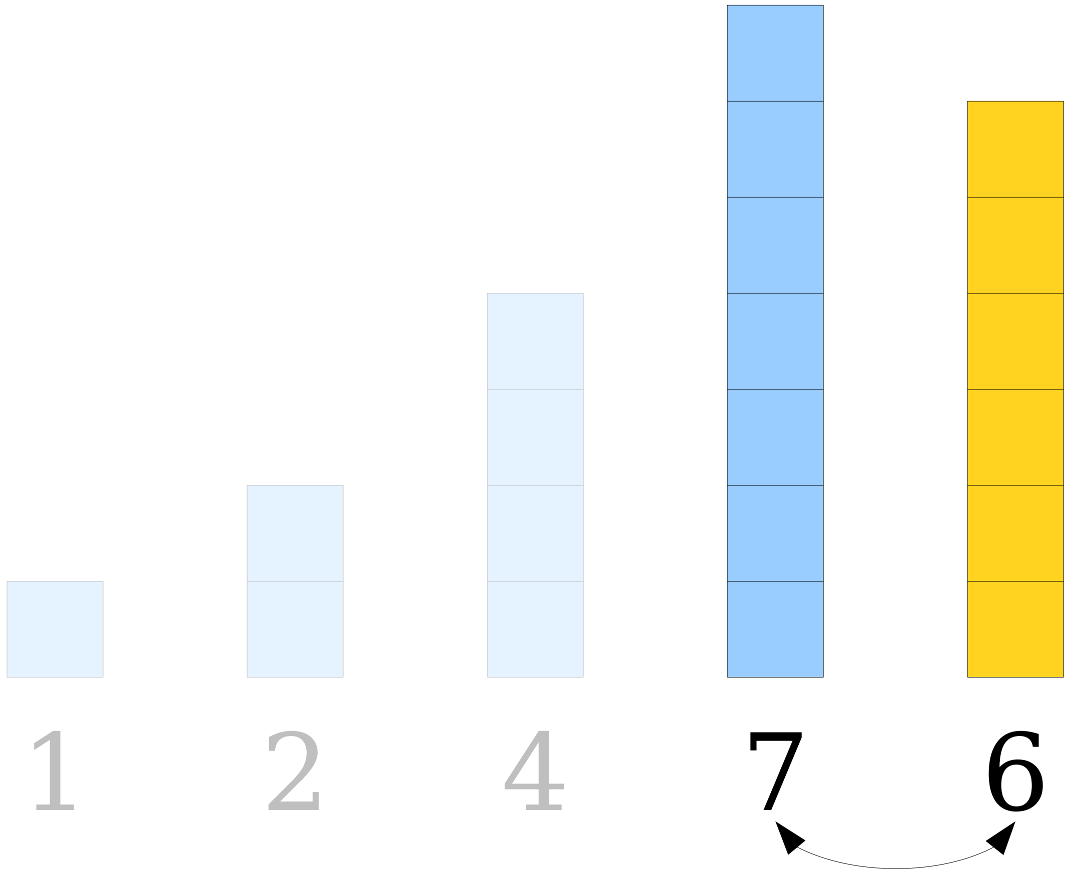


# An Initial Idea: ***Selection Sort***

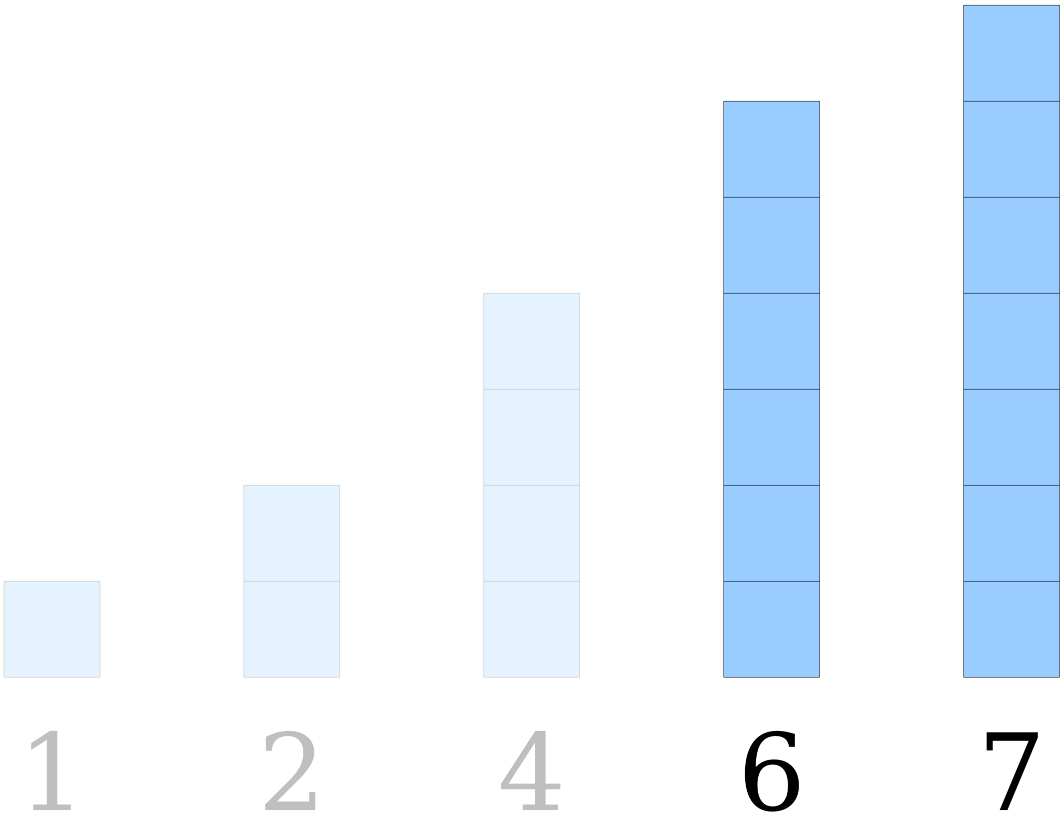



The smallest of these elements needs to go at the front of this group of elements.

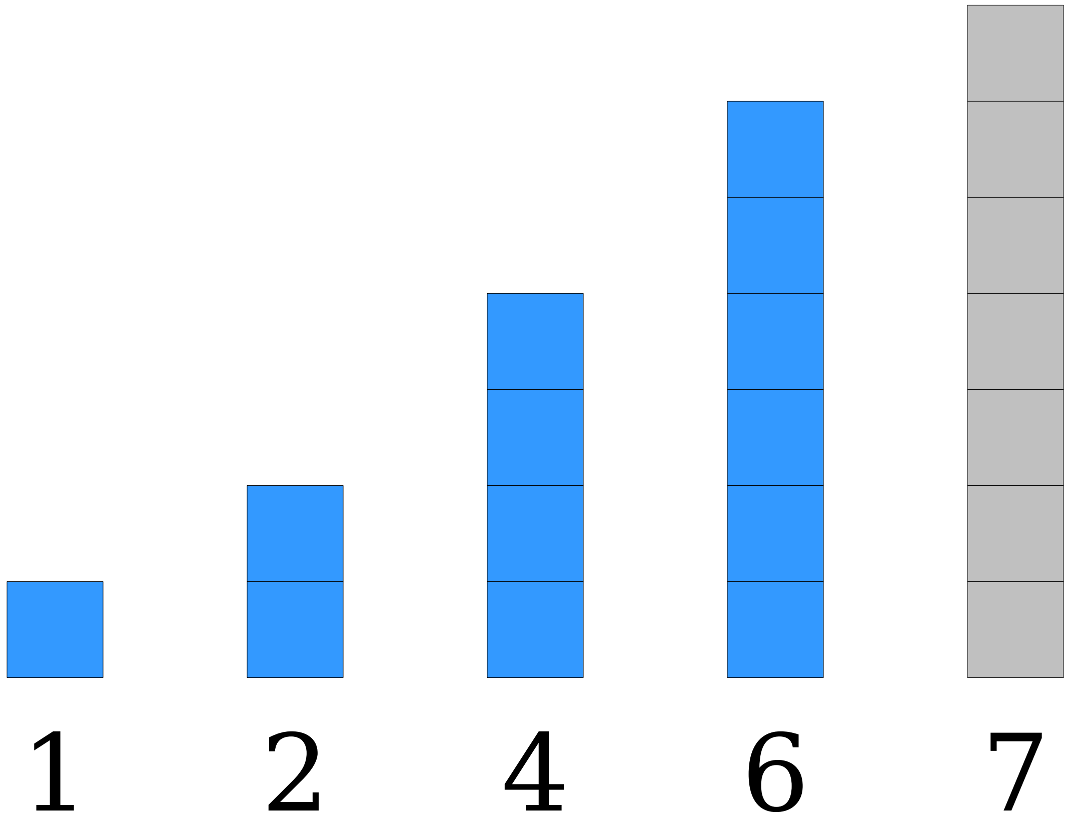
# An Initial Idea: ***Selection Sort***



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



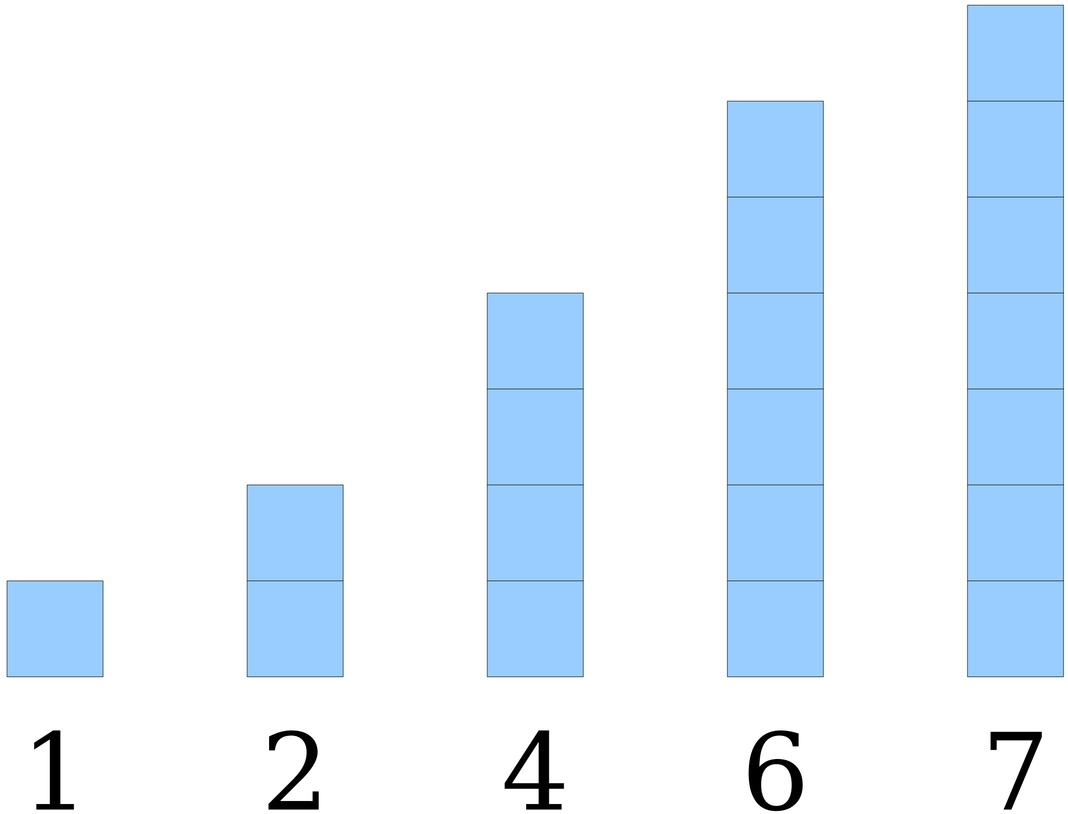
# An Initial Idea: ***Selection Sort***



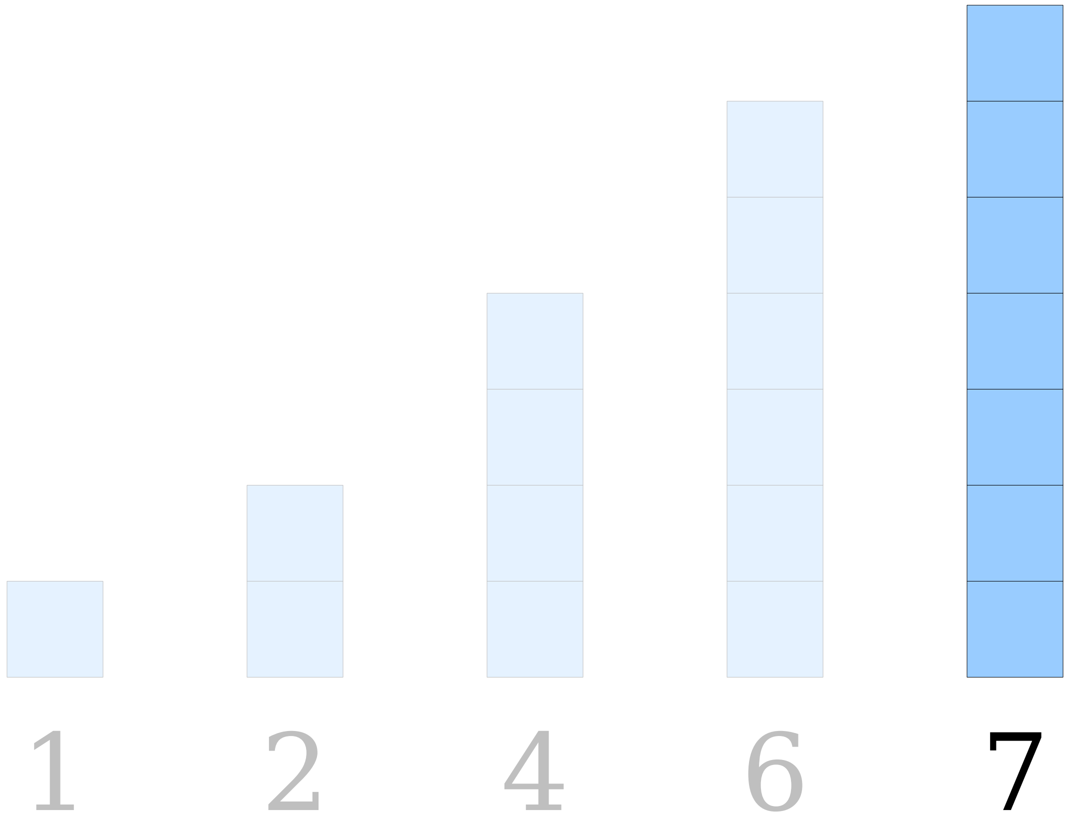

These elements are in the right place now.

The remaining elements are in no particular order.

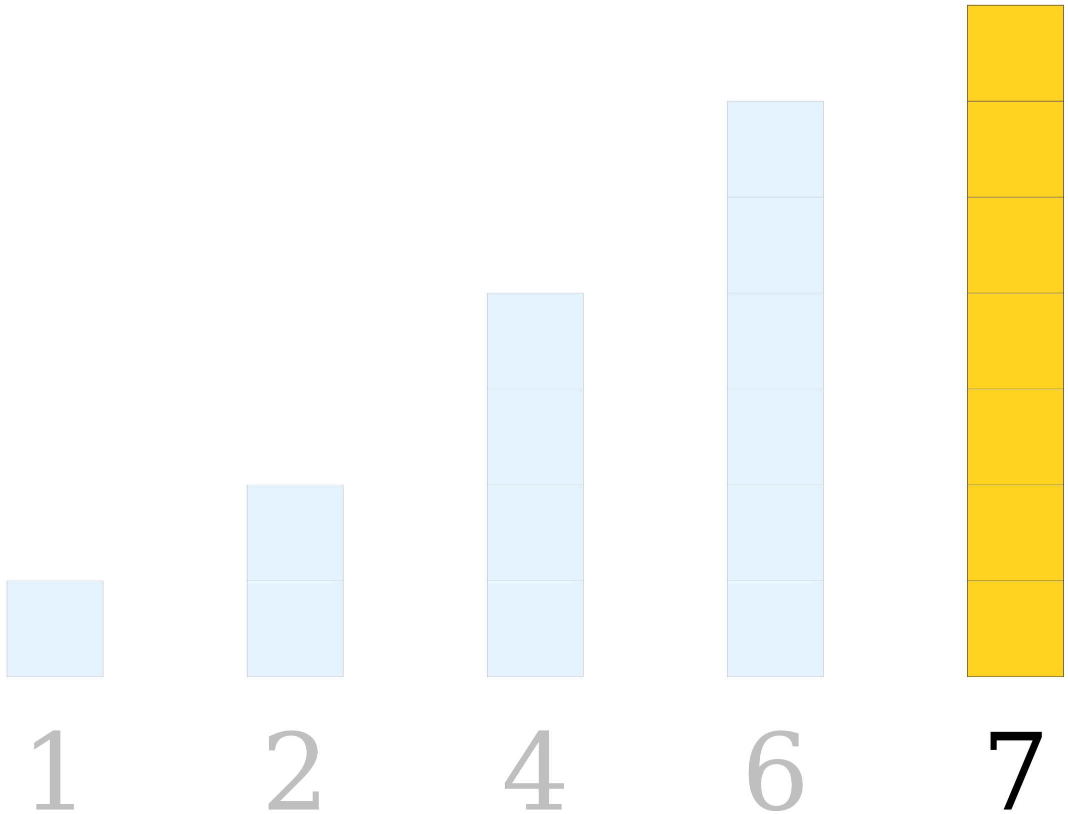
# An Initial Idea: ***Selection Sort***



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

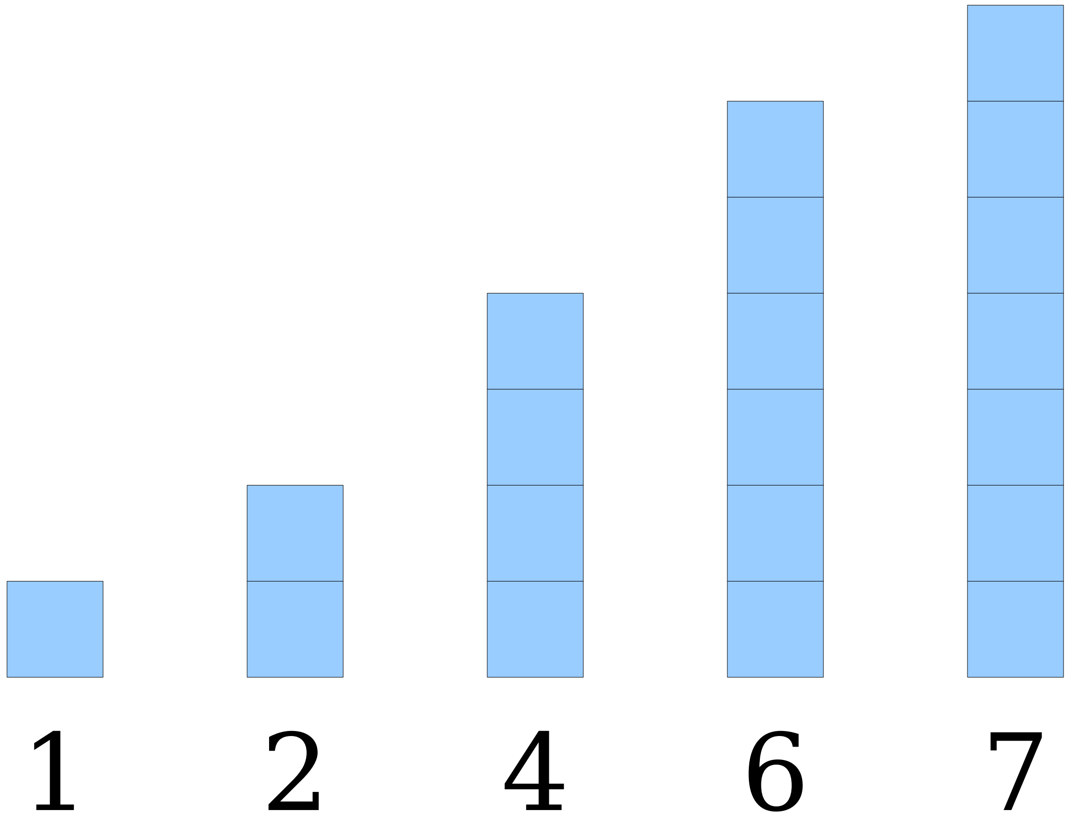


# An Initial Idea: ***Selection Sort***

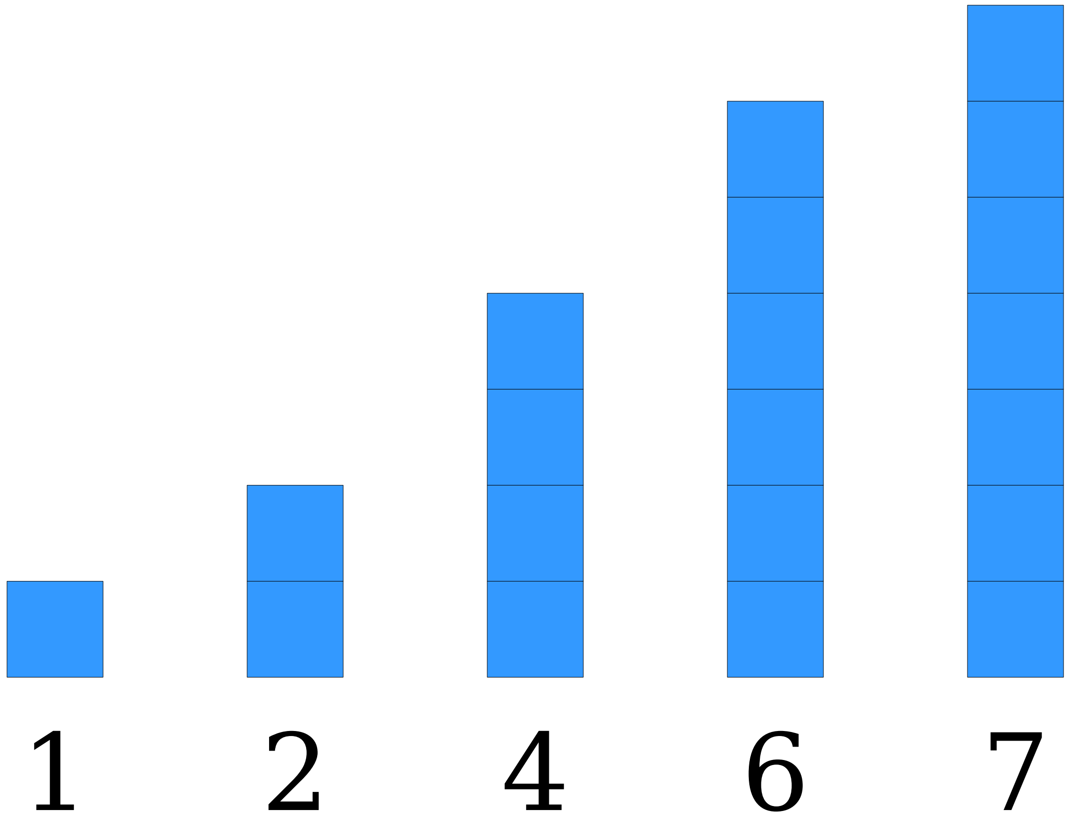



The smallest element from this group needs to go at the front of the group.

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



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




These elements are in the right place now.

# Selection Sort

- Find the smallest element and move it to the first position.
- Find the smallest element of what’s left and move it to the second position.
- Find the smallest element of what’s left and move it to the third position.
- Find the smallest element of what’s left and move it to the fourth position.
- (etc.)

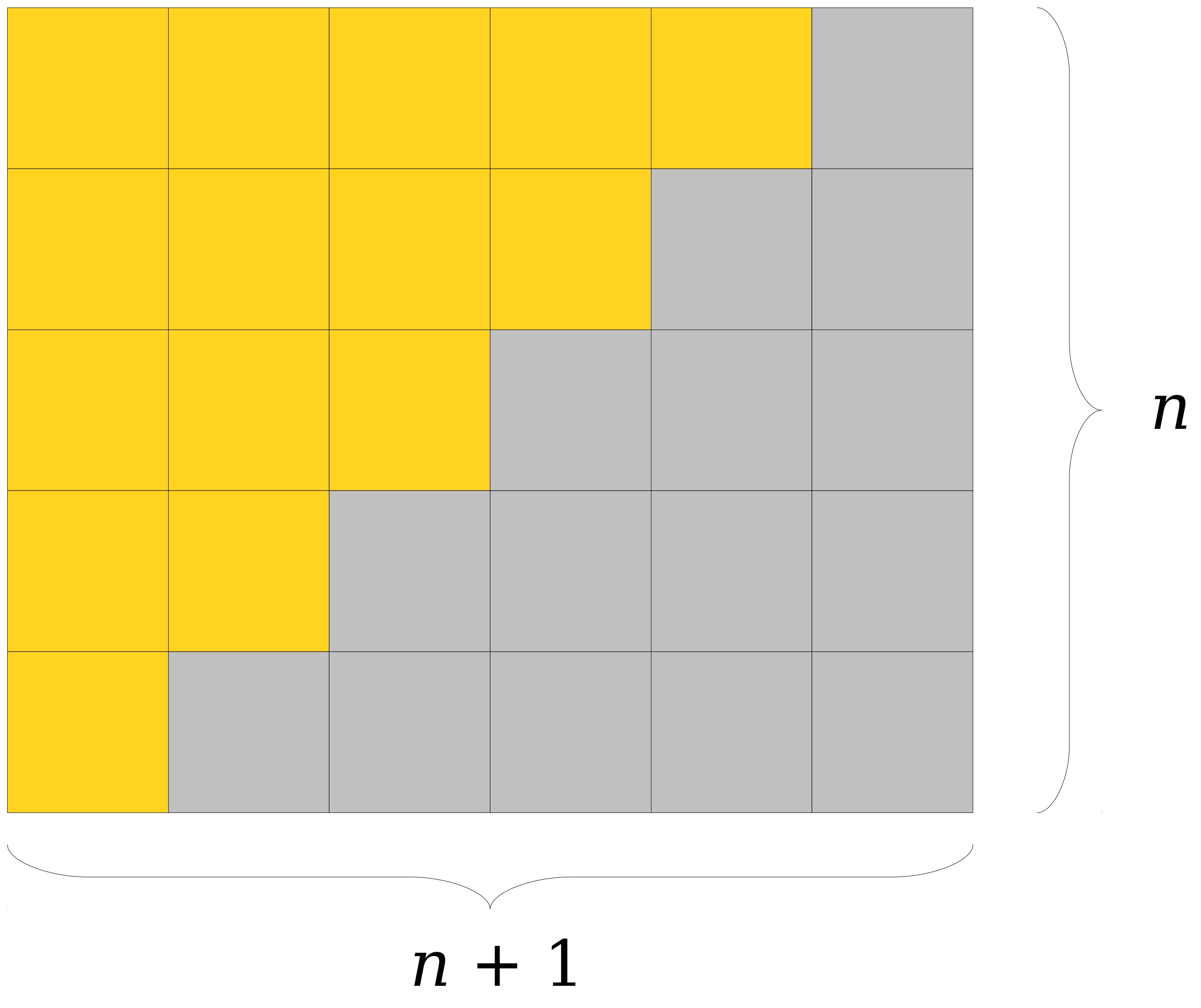
```
/**
 * Sorts the specified vector using the selection sort algorithm.
 *
 * @param elems The elements to sort.
 */
void selectionSort(Vector<int>& elems) {
  for (int index = 0; index < elems.size(); index++) {
    int smallestIndex = indexOfSmallest(elems, index);
    swap(elems[index], elems[smallestIndex]);
  }
}

/**
 * Given a vector and a starting point, returns the index of the smallest
 * element in that vector at or after the starting point
 *
 * @param elems The elements in question.
 * @param startPoint The starting index in the vector.
 * @return The index of the smallest element at or after that point
 */
int indexOfSmallest(const Vector<int>& elems, int startPoint) {
  int smallestIndex = startPoint;
  for (int i = startPoint + 1; i < elems.size(); i++) {
    if (elems[i] < elems[smallestIndex]) {
      smallestIndex = i;
    }
  }
  return smallestIndex;
}
```

# Analyzing Selection Sort

- How much work do we do for selection sort?
- To find the smallest value, we need to look at all $n$ array elements.
- To find the second-smallest value, we need to look at $n - 1$ array elements.
- To find the third-smallest value, we need to look at $n - 2$ array elements.
- Work is $n + (n - 1) + (n - 2) + \ldots + 1$.

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



# The Complexity of Selection Sort

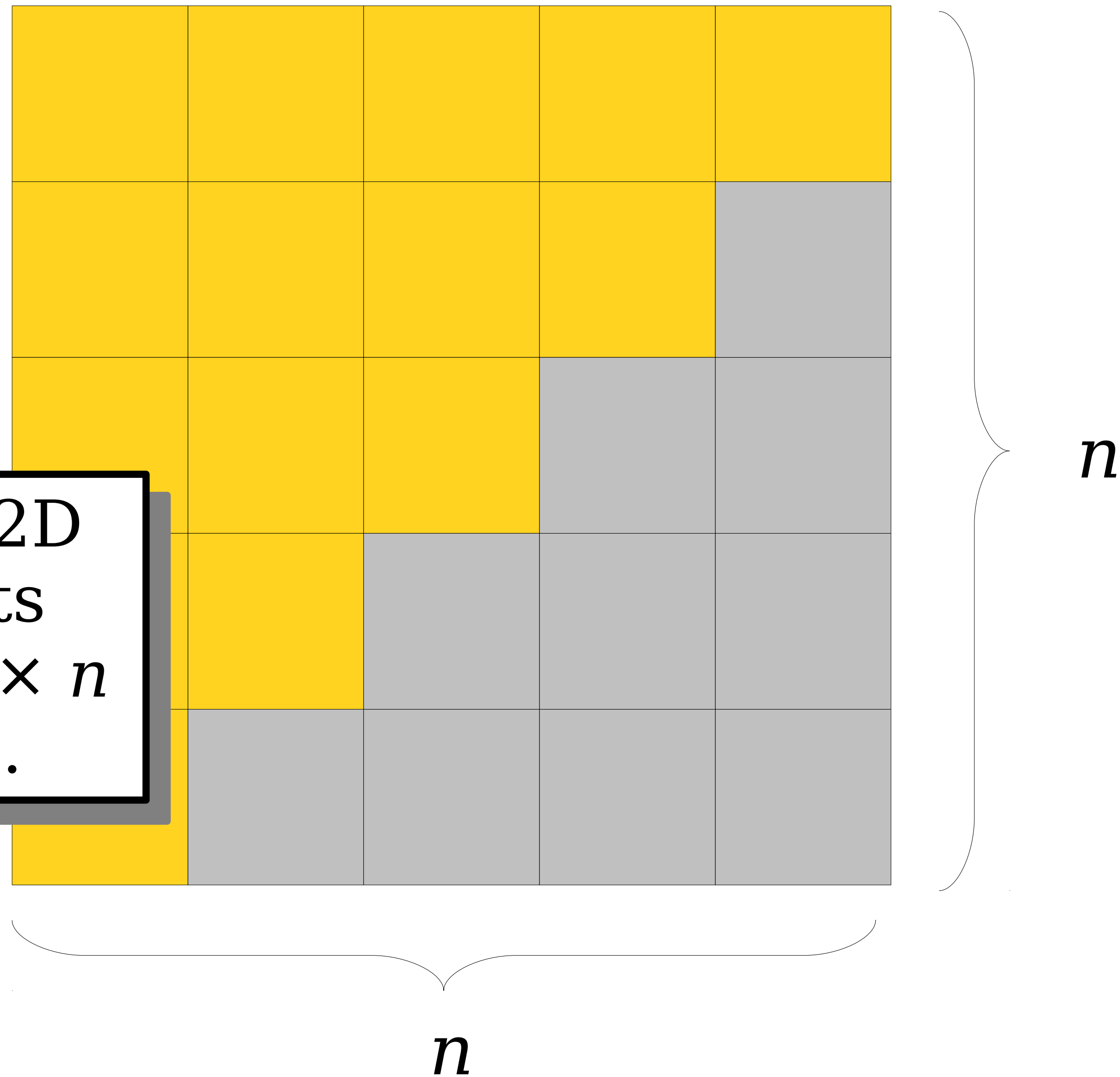
$$
\begin{aligned}
& O(n\,(n + 1)\,/\,2) \\
= \; & O(n\,(n + 1)) \\
= \; & O(n^2 + n) \\
= \; & O(n^2)
\end{aligned}
$$

Big-O notation ignores constant factors. (Think "area of a circle" and "area of a square.")

Big-O notation ignores low-order terms. Think "cost of making *n* widgets."

So selection sort runs in time $\mathbf{O(n^2)}$.

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




The area of a 2D figure that fits snugly in an $n \times n$ box is $\mathbf{O(n^2)}$.

Our theory predicts that the runtime of selection sort is $O(n^2)$.

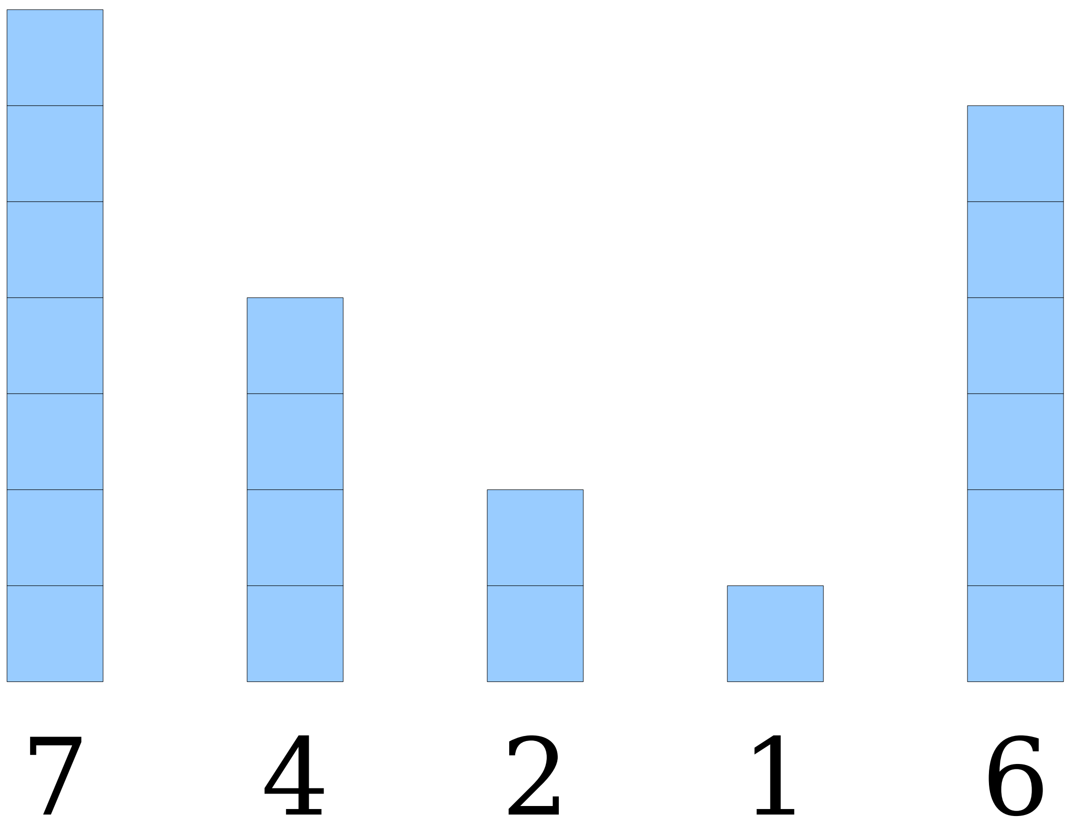
Does that match what we see in practice?

What should we expect to see when we look at a runtime plot?

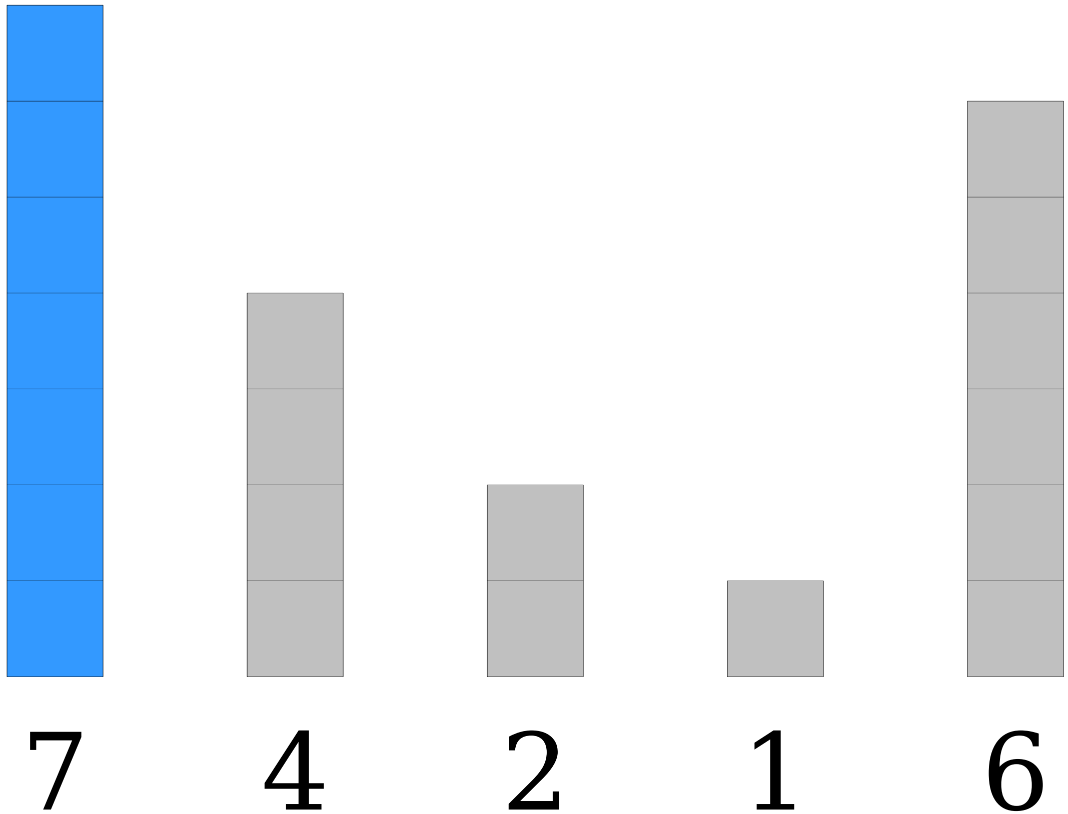
# Another Sorting Algorithm

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

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



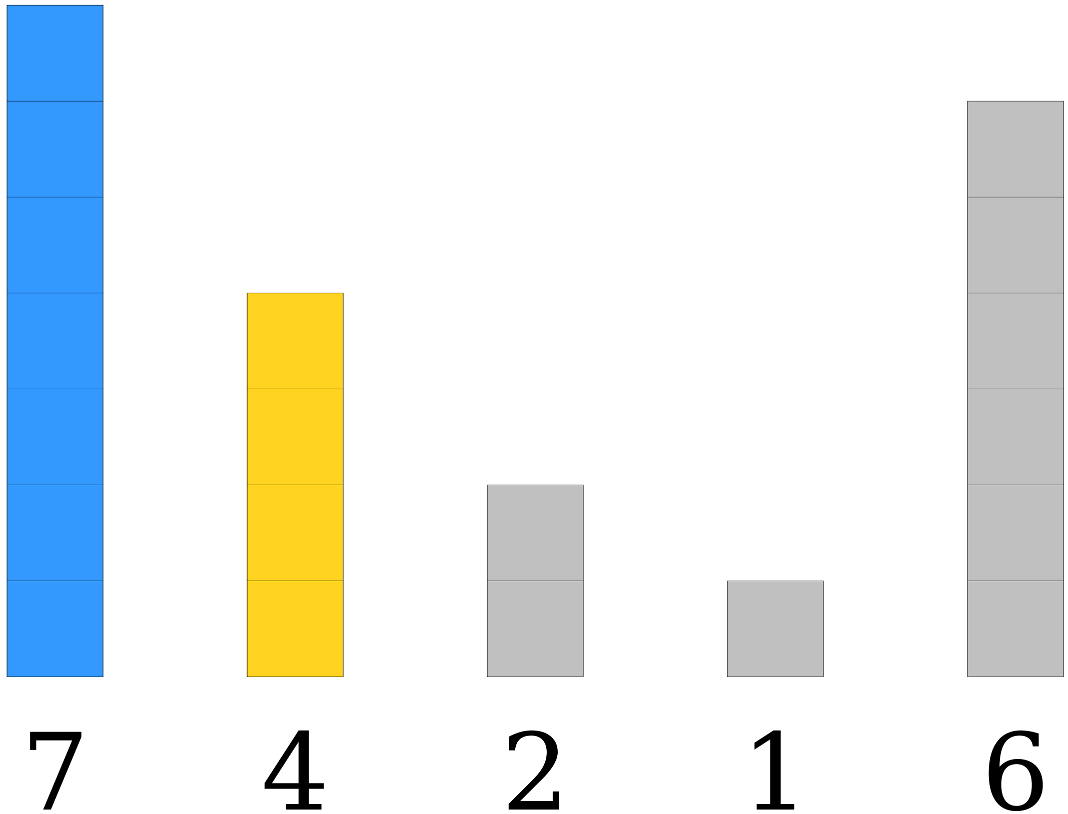
# Our Next Idea: ***Insertion Sort***




This sequence in blue, taken in isolation, is in sorted order.

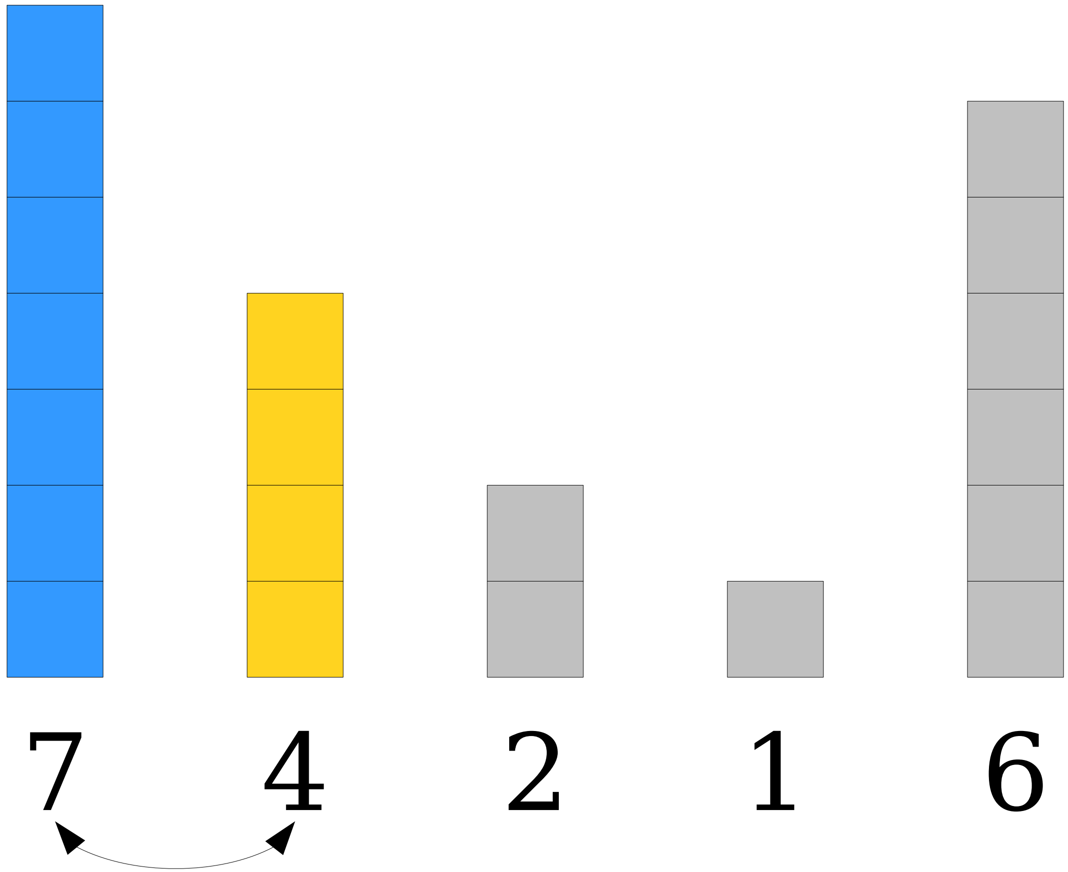
This sequence in gray is in no particular order.

# Our Next Idea: *Insertion Sort*

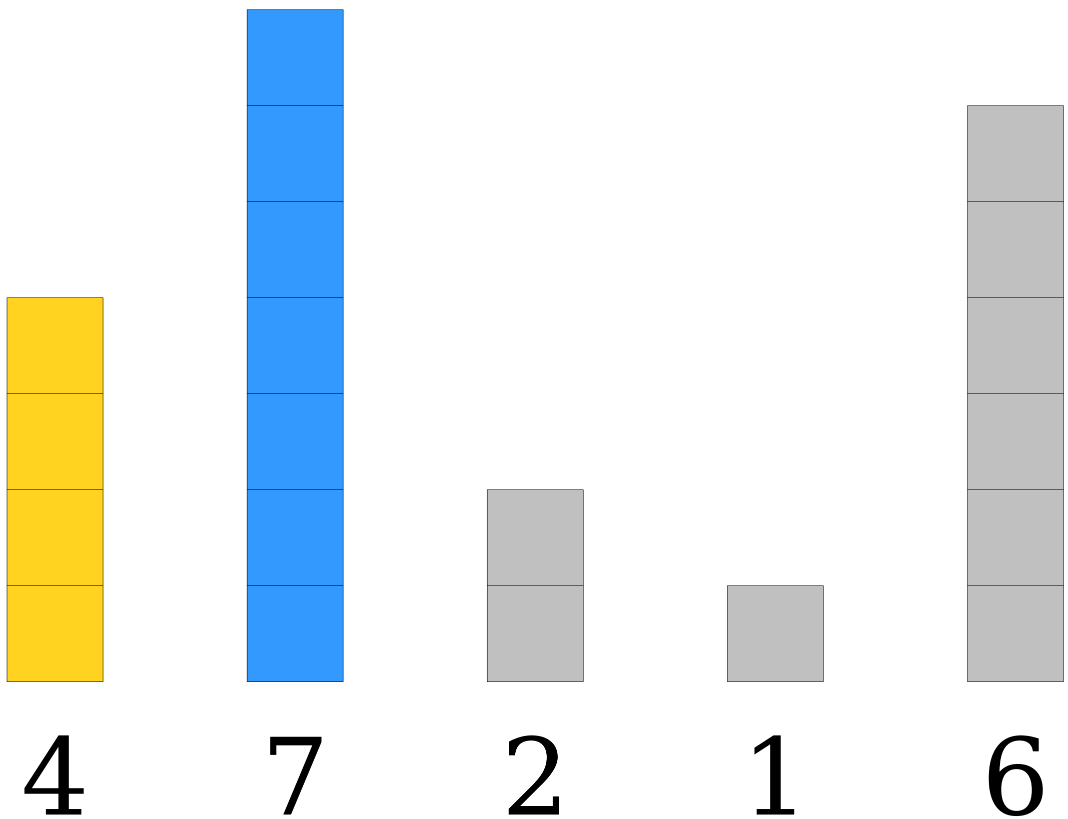



***Insert*** this element into the blue sequence, making the blue sequence one element longer.

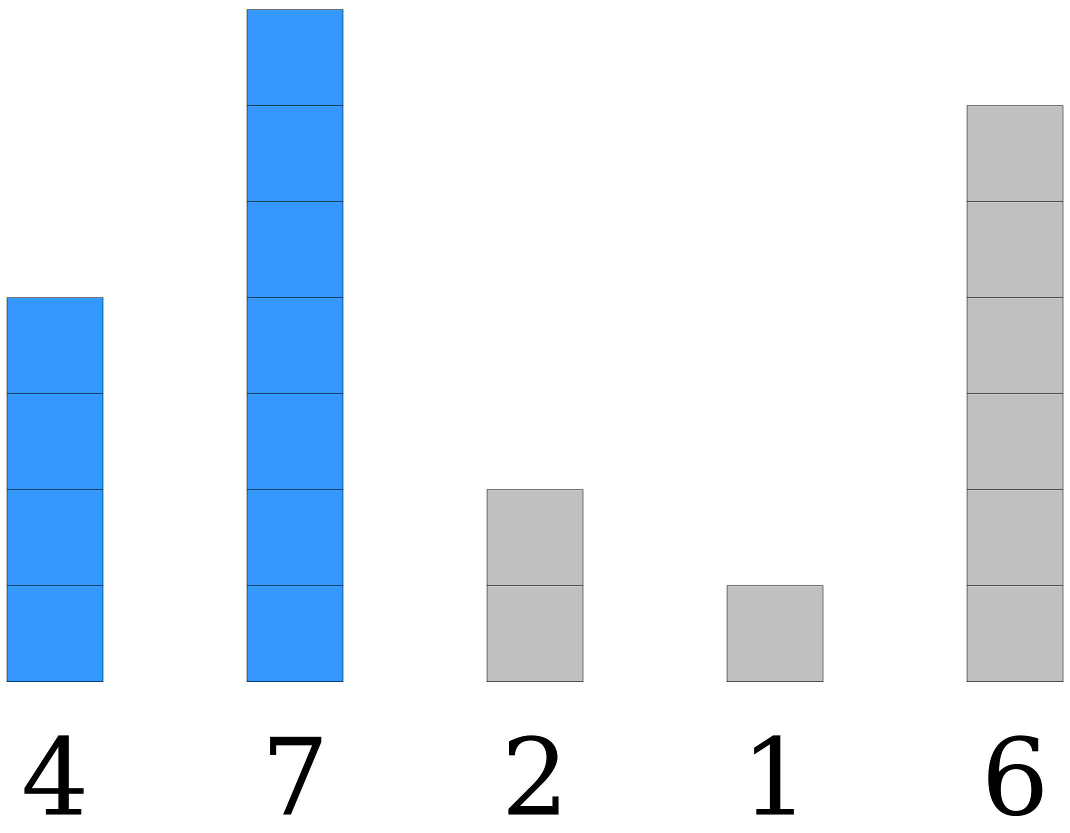
# Our Next Idea: ***Insertion Sort***



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



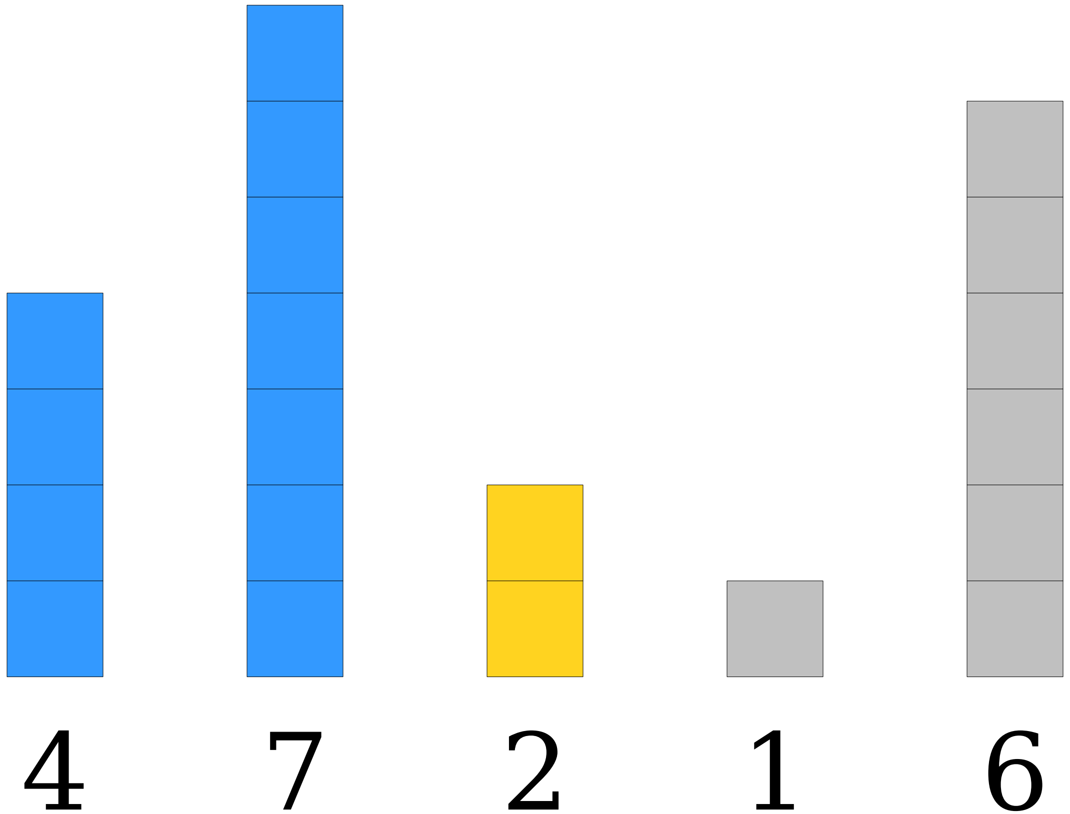
# Our Next Idea: ***Insertion Sort***




This sequence in blue, taken in isolation, is in sorted order.

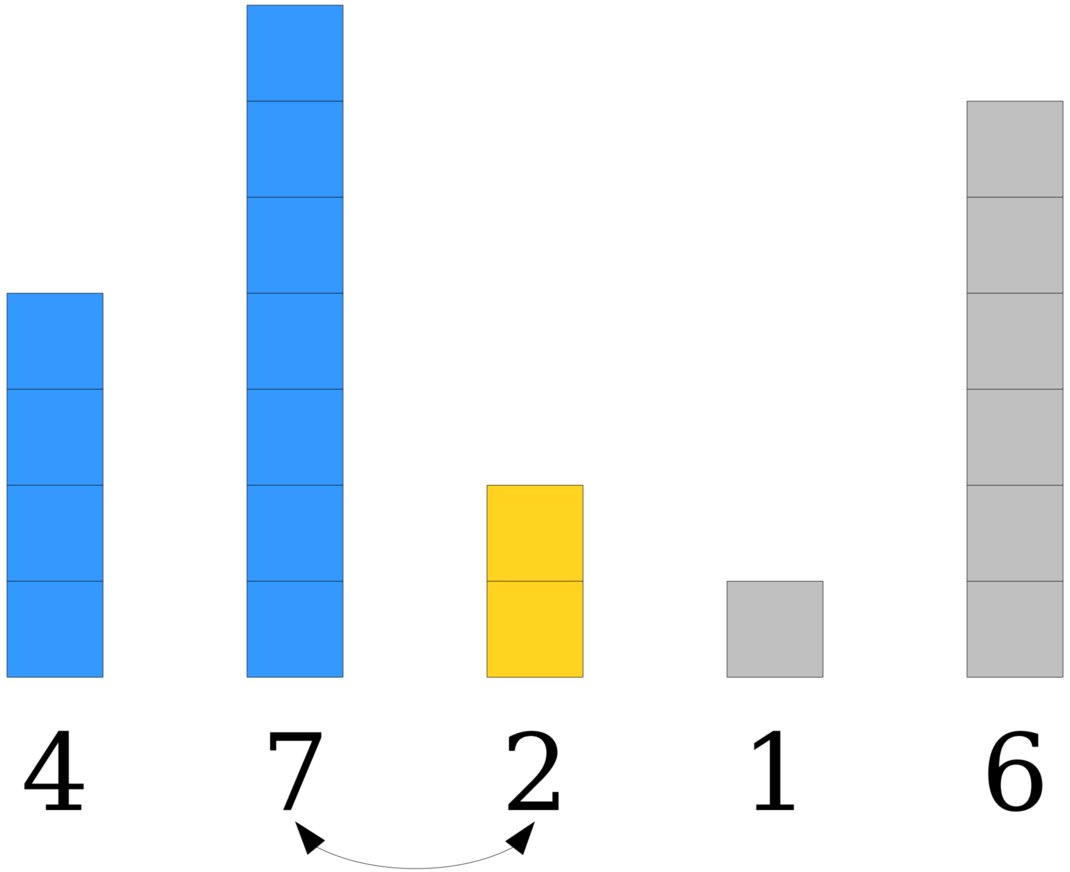
This sequence in gray is in no particular order.

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

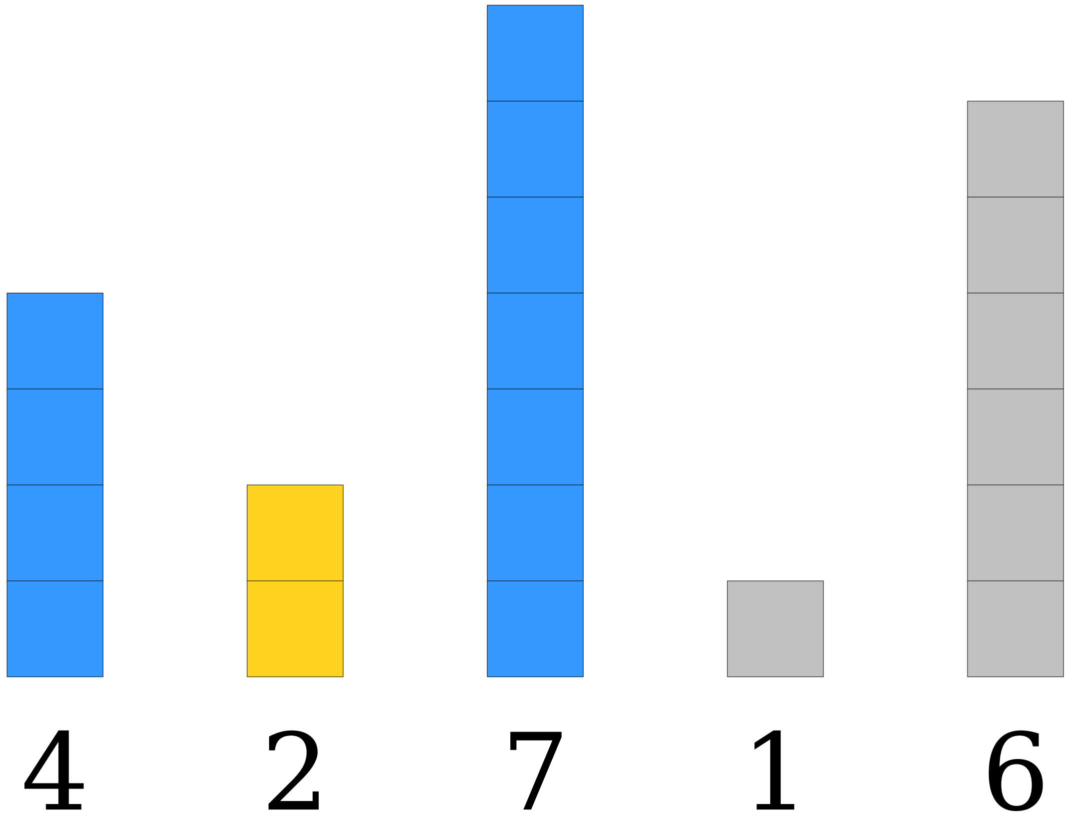



***Insert*** this element into the blue sequence, making the blue sequence one element longer.

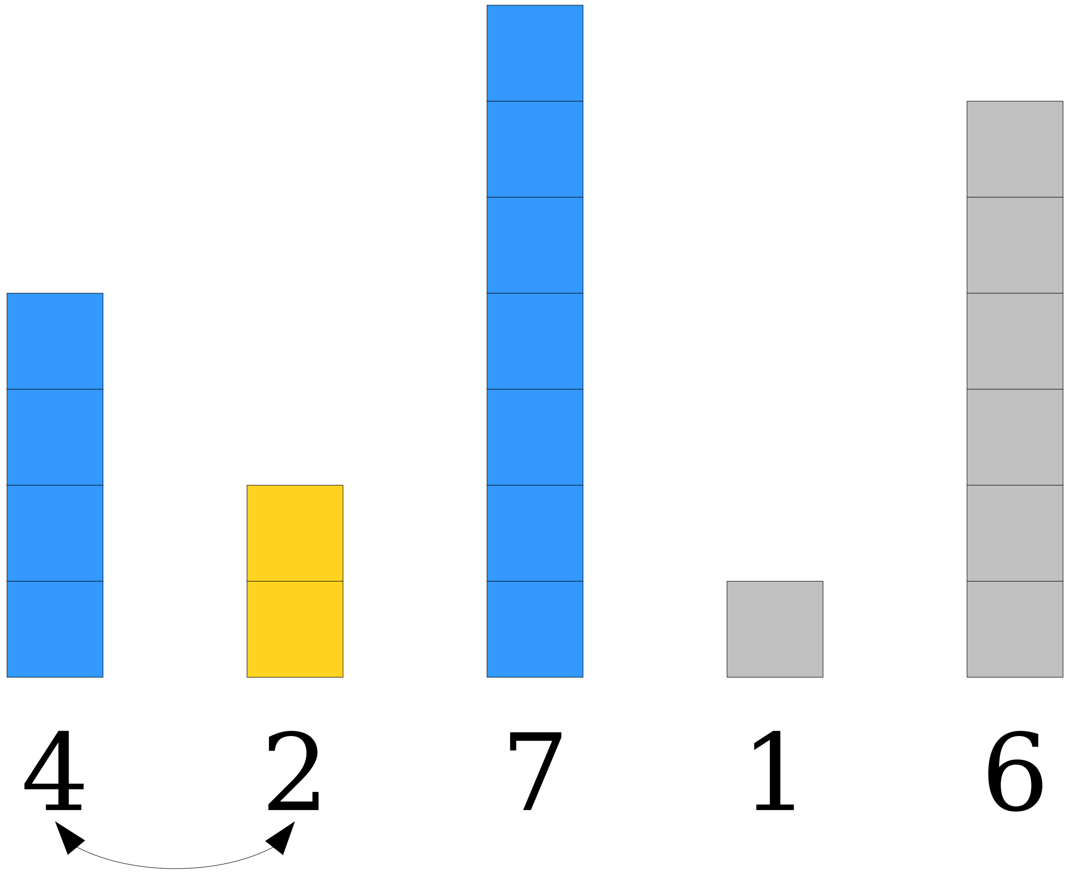
# Our Next Idea: ***Insertion Sort***



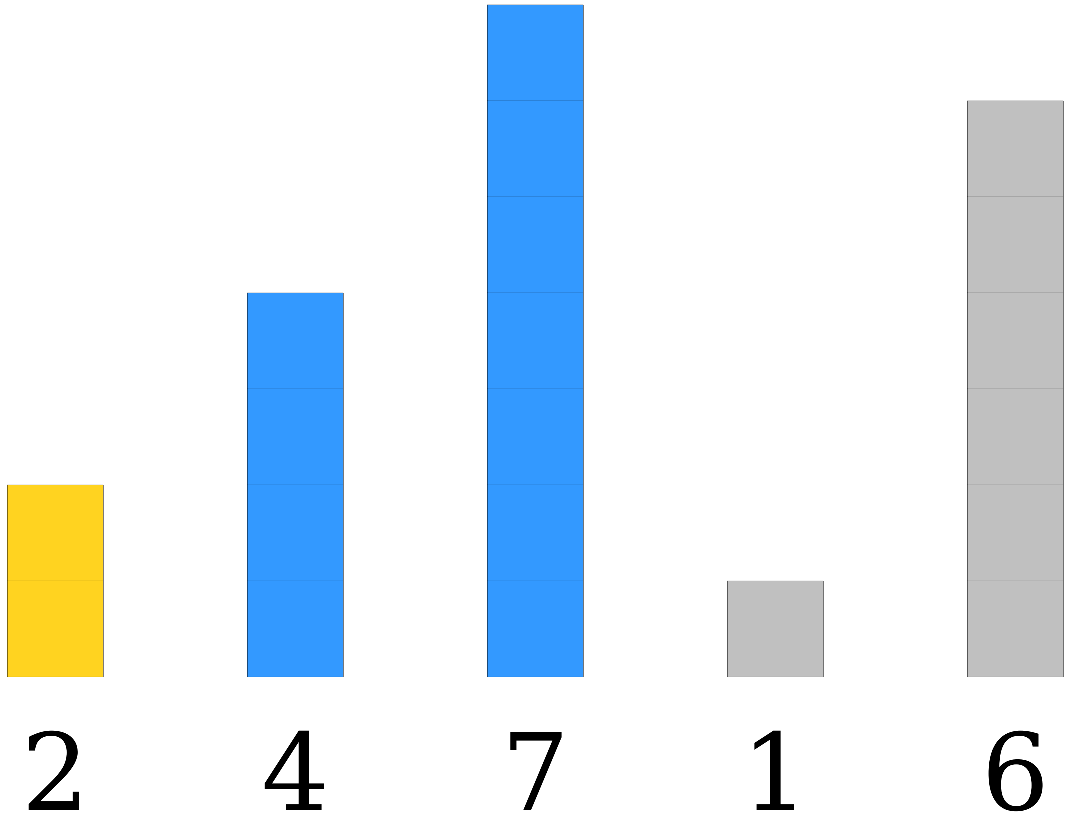
# Our Next Idea: ***Insertion Sort***



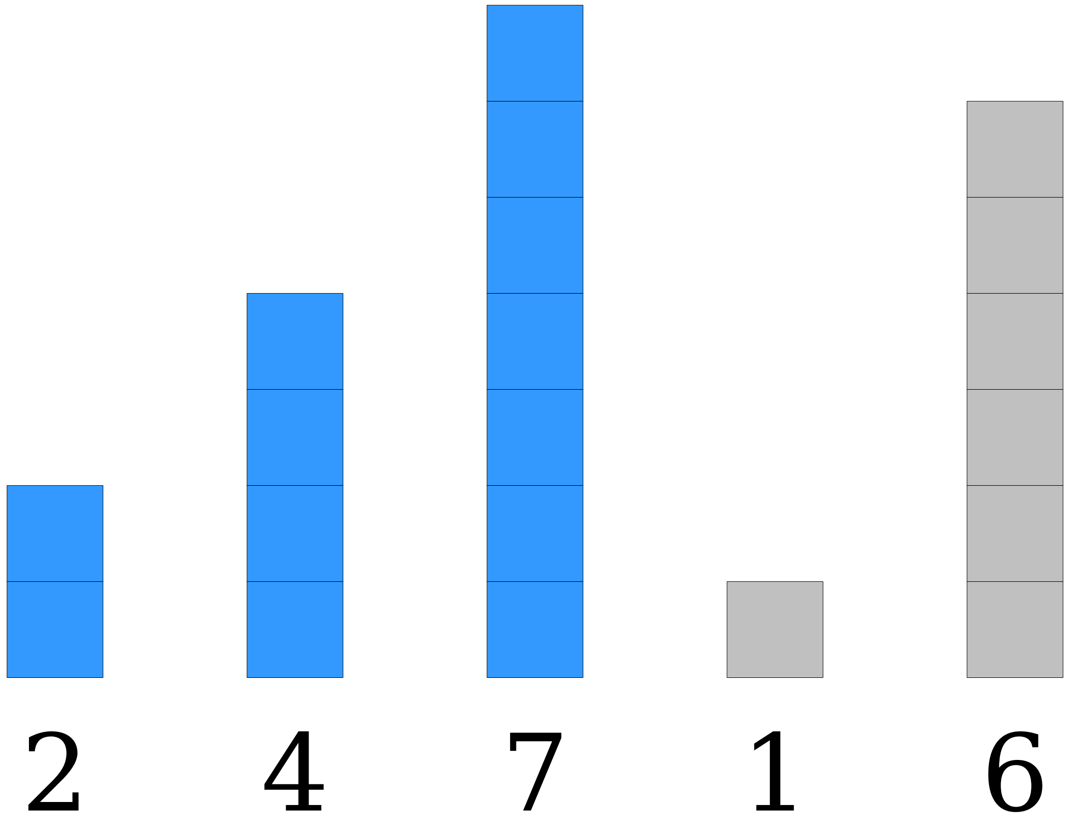
# Our Next Idea: ***Insertion Sort***



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



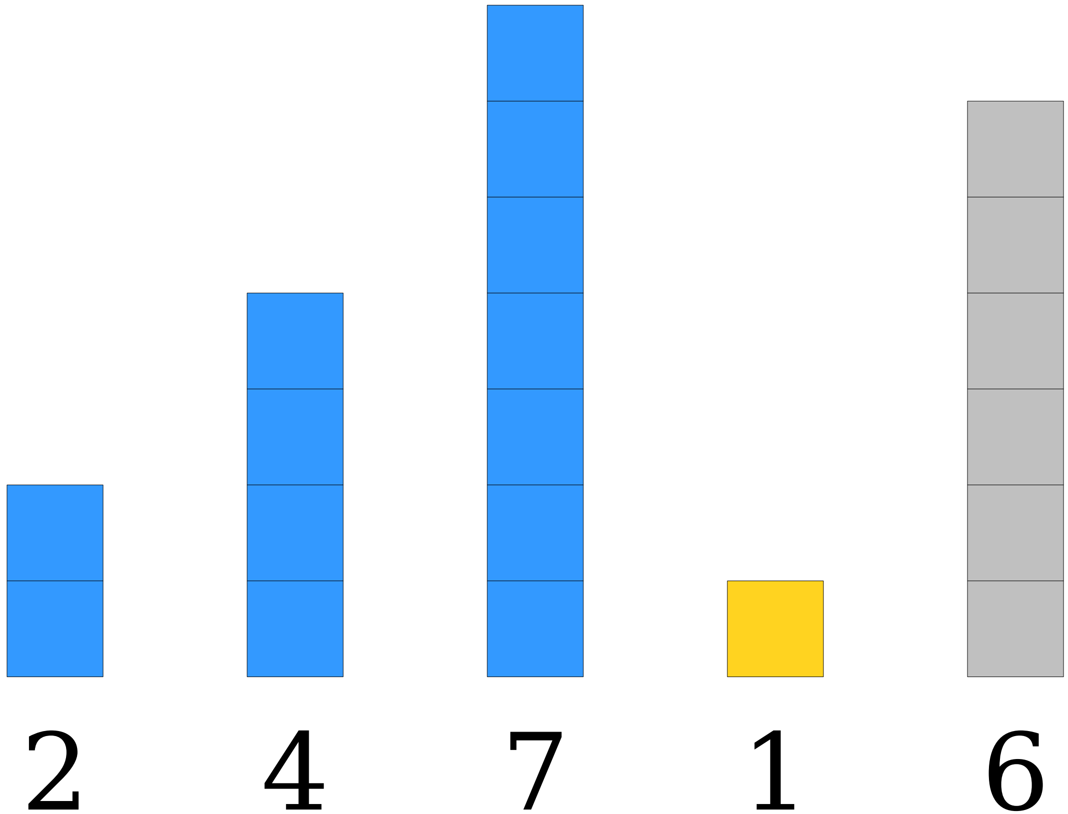
# Our Next Idea: ***Insertion Sort***




This sequence in blue, taken in isolation, is in sorted order.

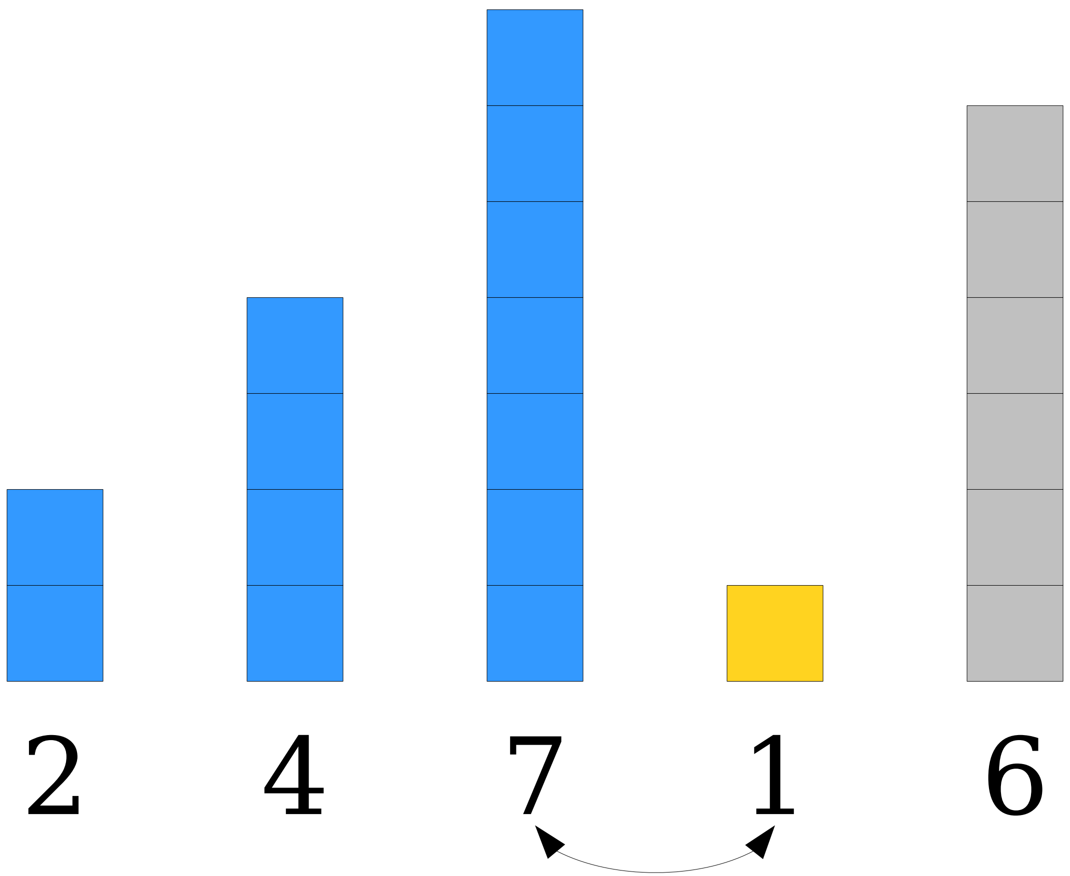
This sequence in gray is in no particular order.

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

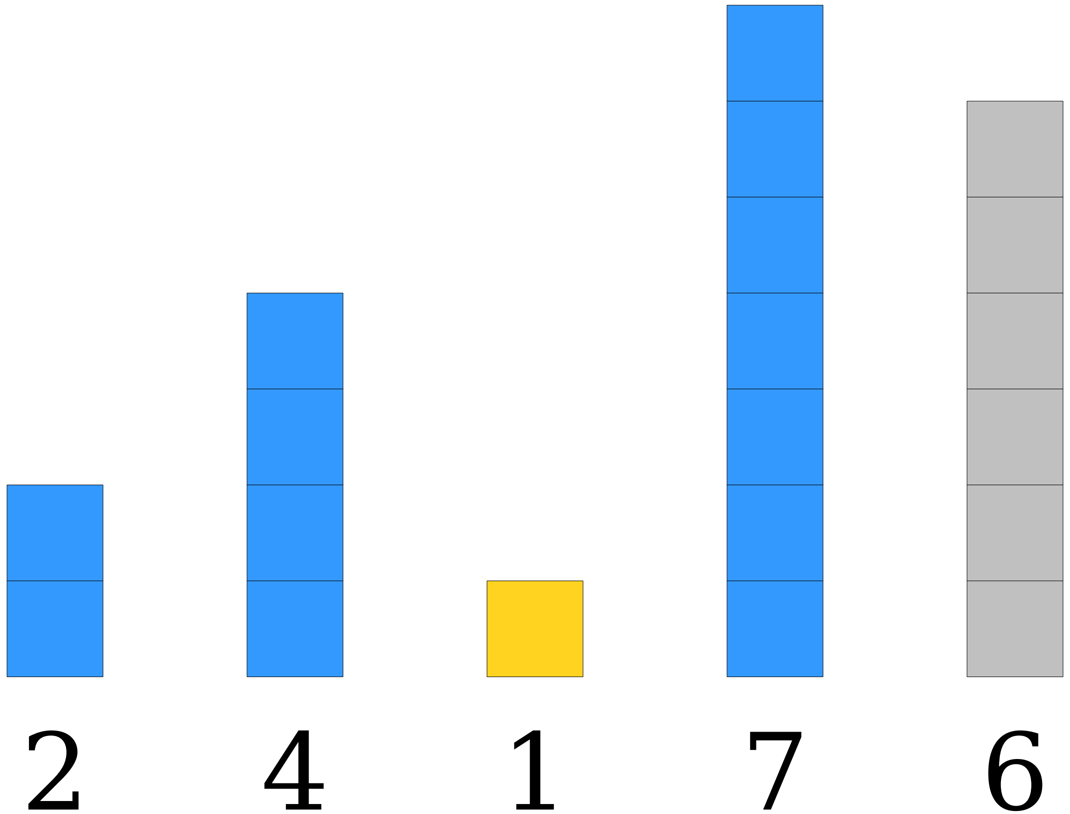



***Insert*** this element into the blue sequence, making the blue sequence one element longer.

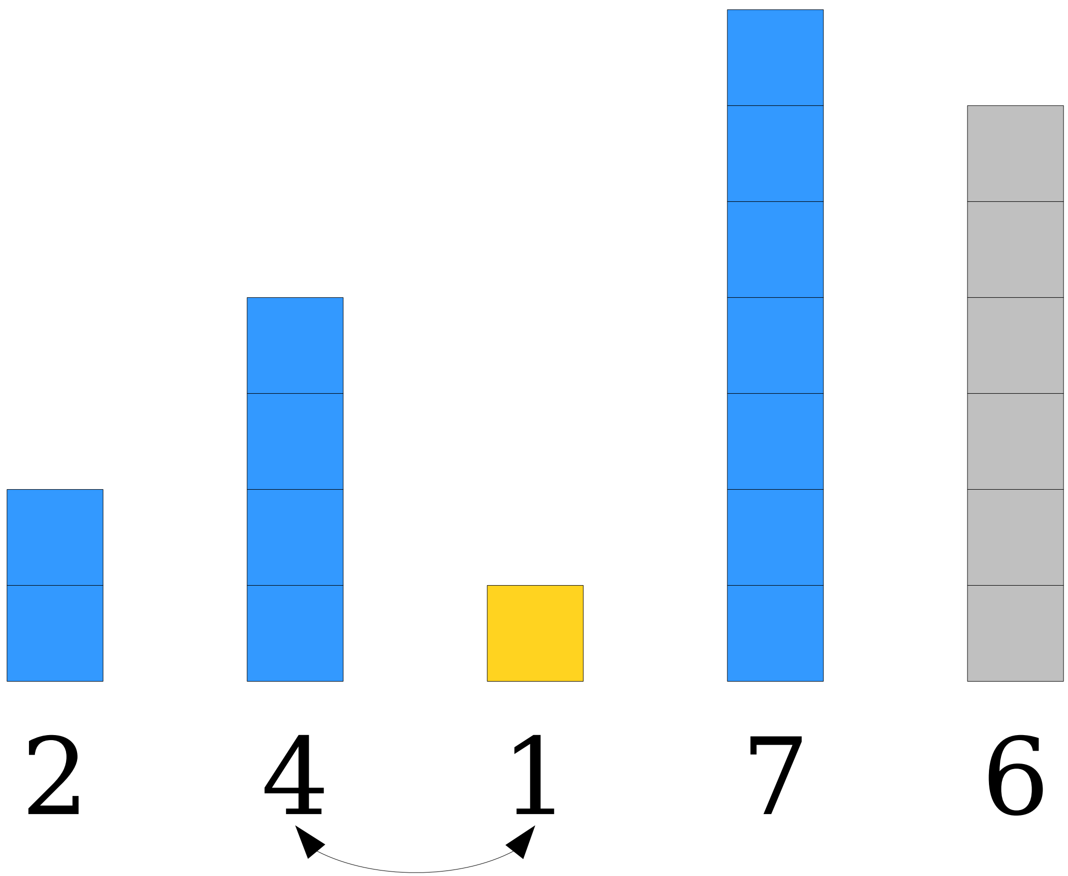
# Our Next Idea: ***Insertion Sort***



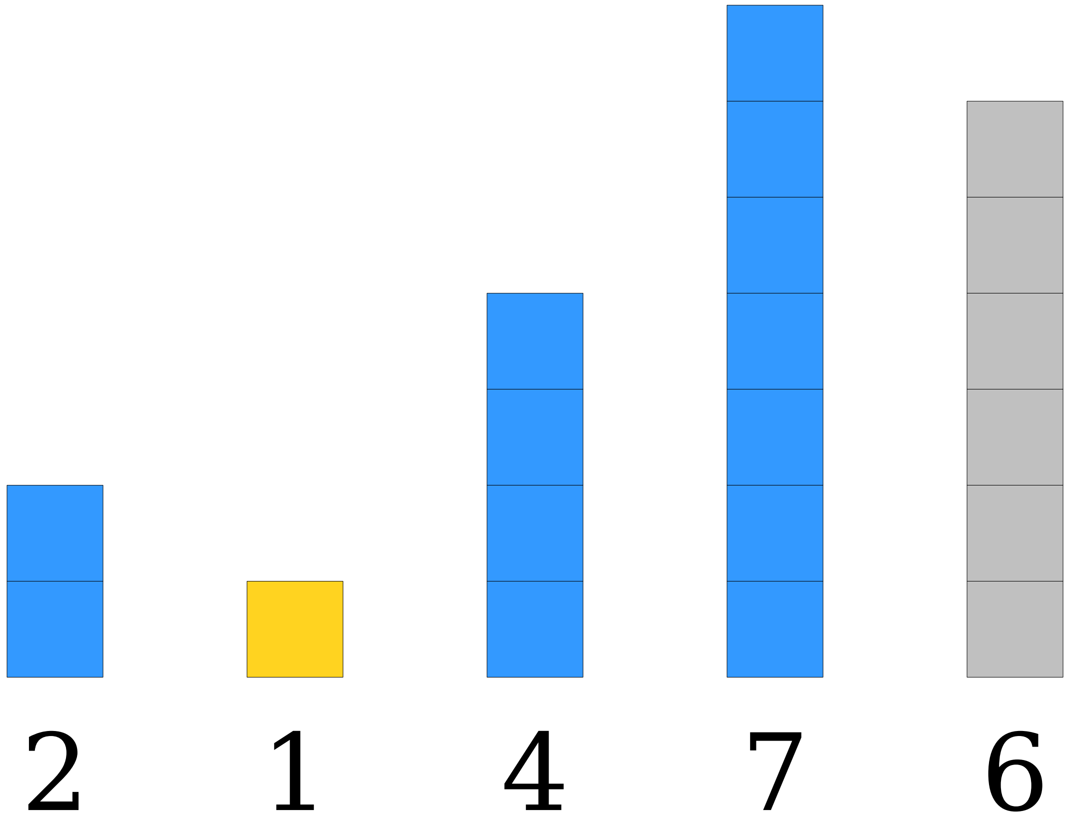
# Our Next Idea: ***Insertion Sort***



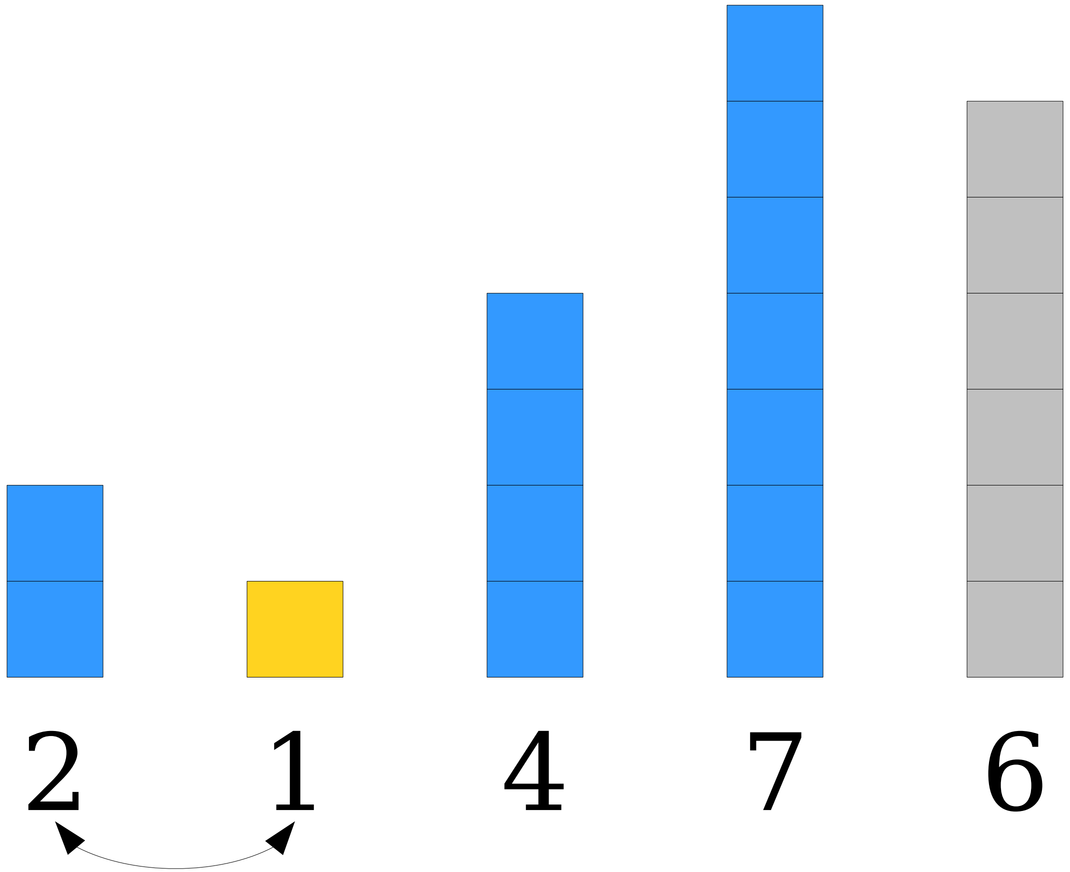
# Our Next Idea: ***Insertion Sort***



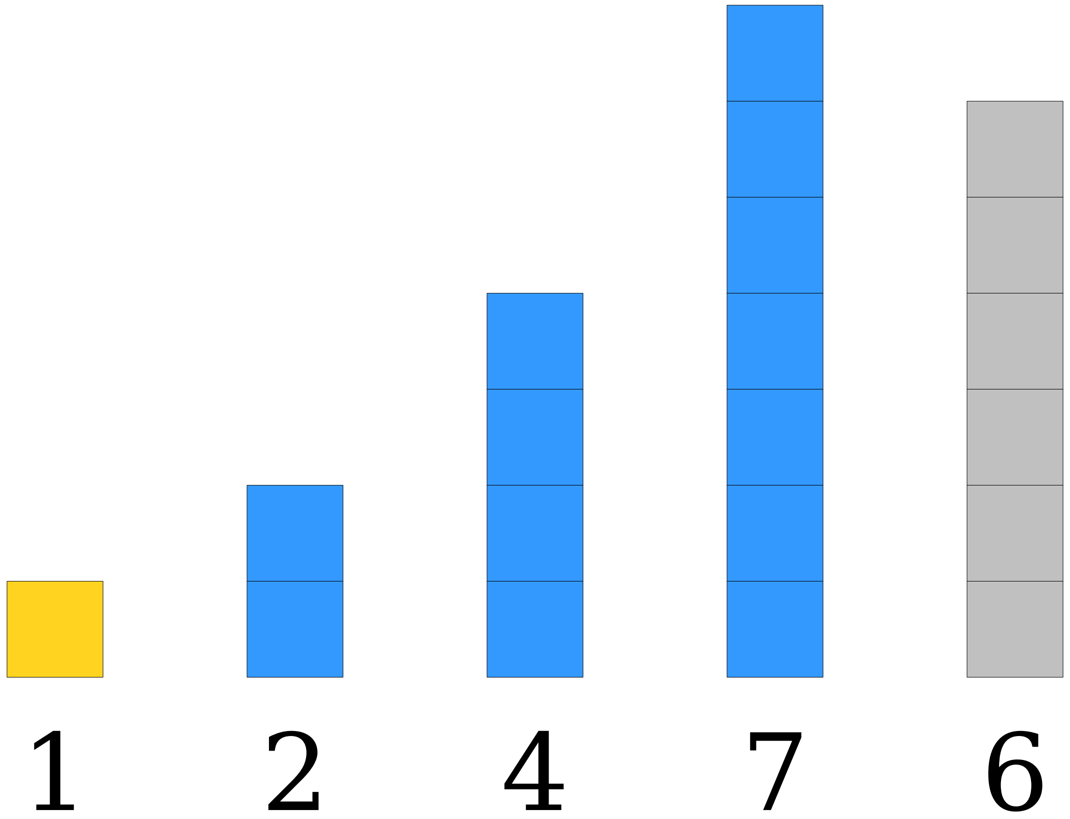
# Our Next Idea: ***Insertion Sort***



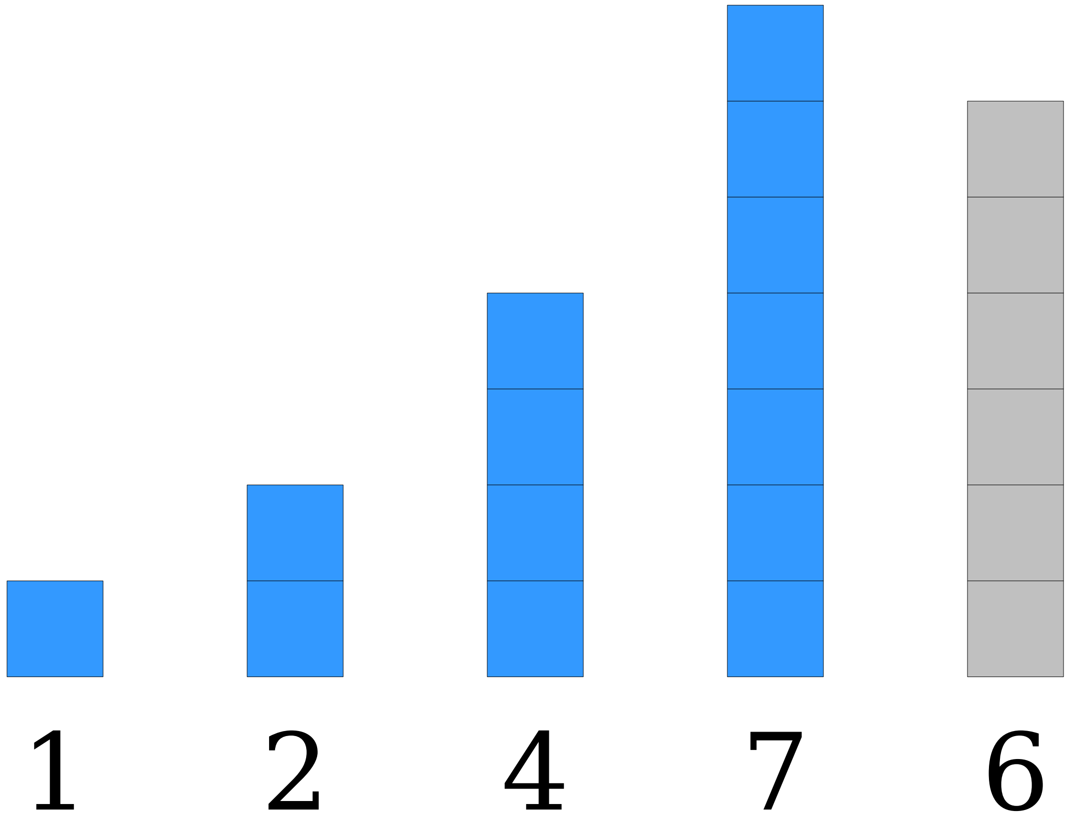
# Our Next Idea: ***Insertion Sort***



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



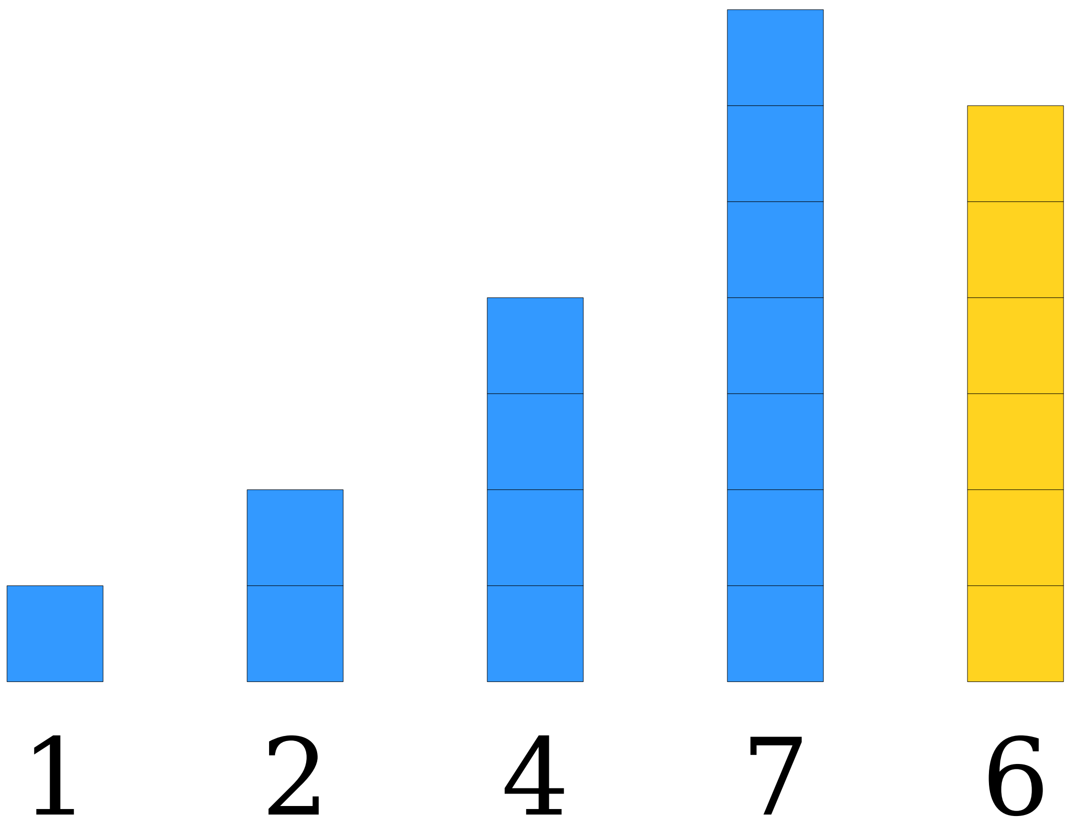
# Our Next Idea: ***Insertion Sort***




This sequence in blue, taken in isolation, is in sorted order.

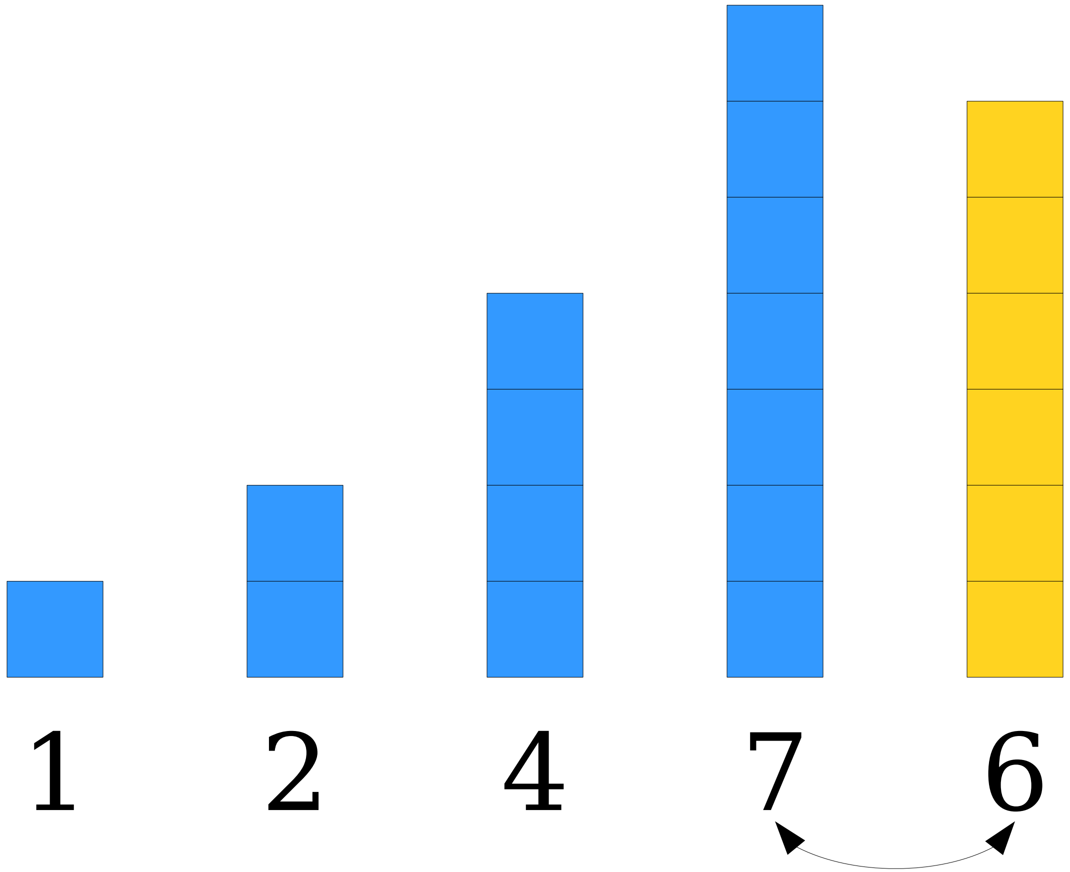
This sequence in gray is in no particular order.

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

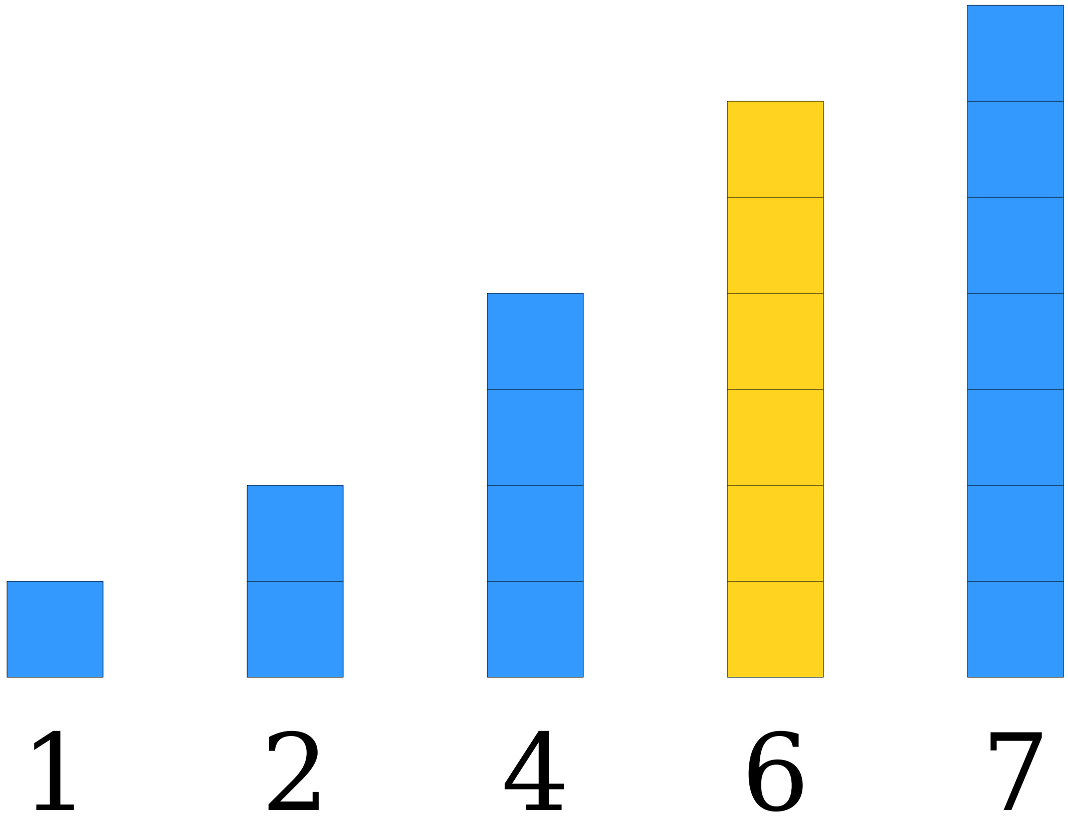



***Insert*** this element into the blue sequence, making the blue sequence one element longer.

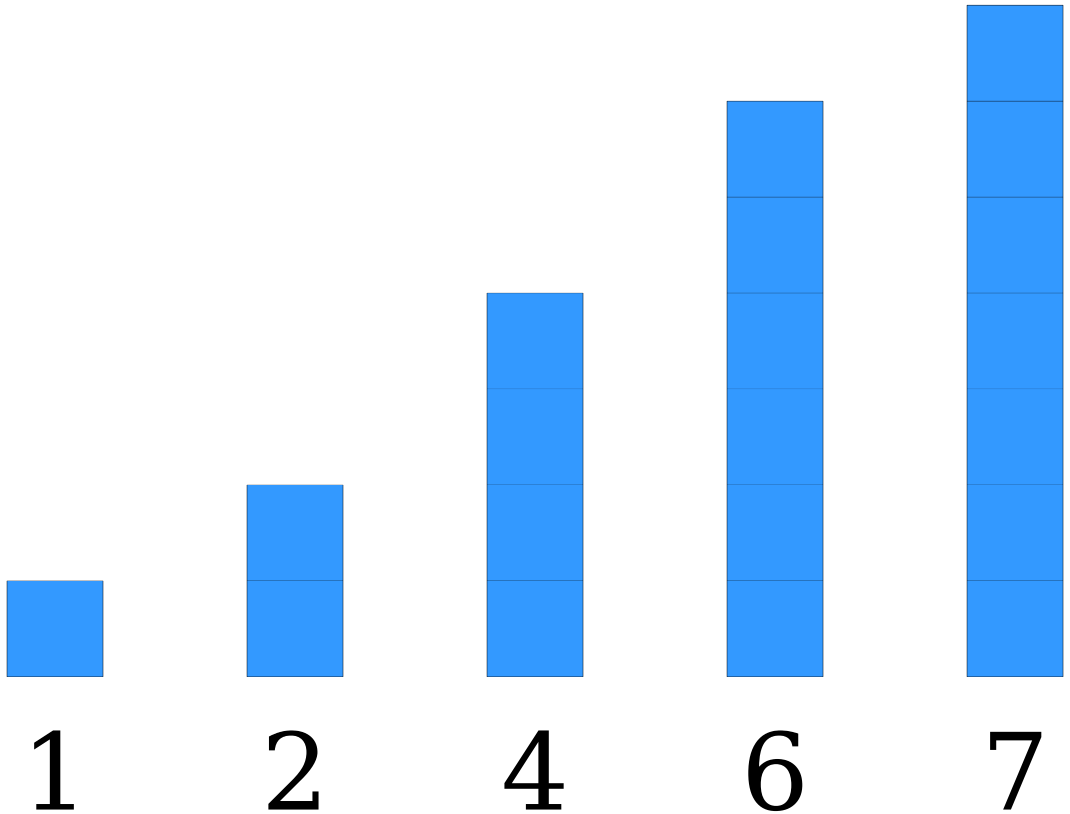
# Our Next Idea: ***Insertion Sort***



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



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




This sequence in blue, taken in isolation, is in sorted order.

There are no more gray elements, so the sequence is sorted!

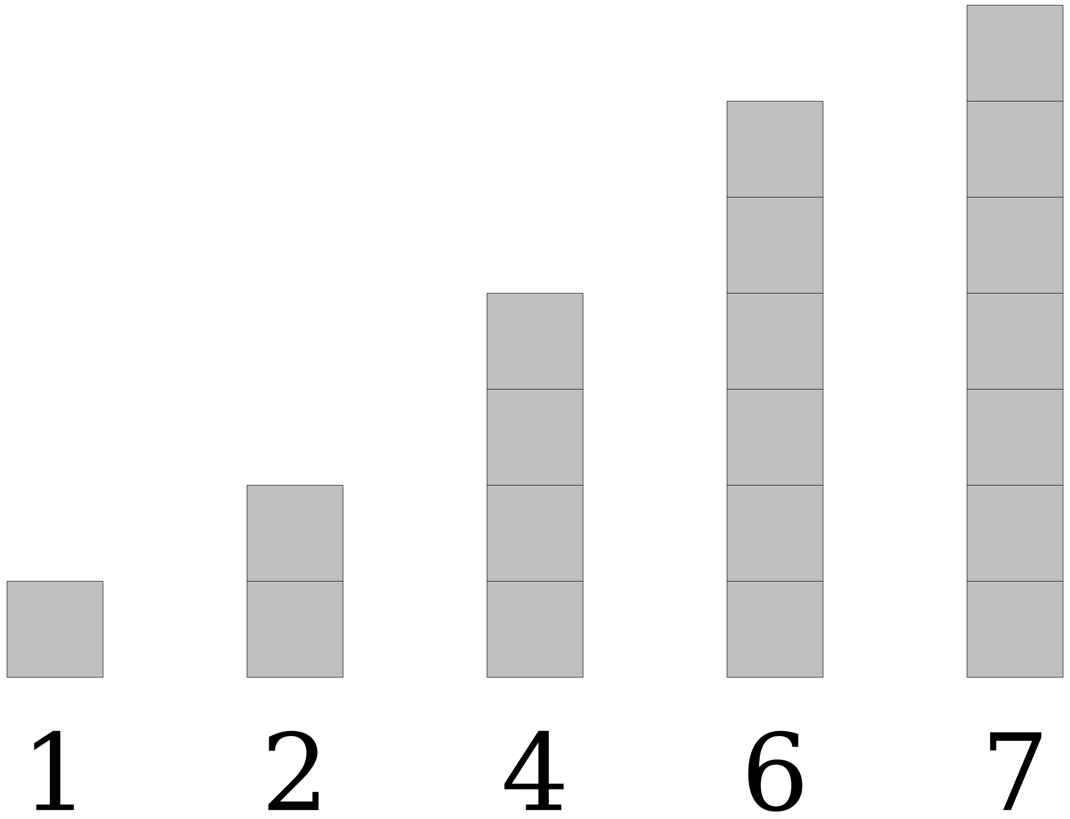
# Insertion Sort

- Repeatedly *insert* an element into a sorted sequence at the front of the array.
- To *insert* an element, swap it backwards until either
  - (1) it's bigger than the element before it, or
  - (2) it's at the front of the array.

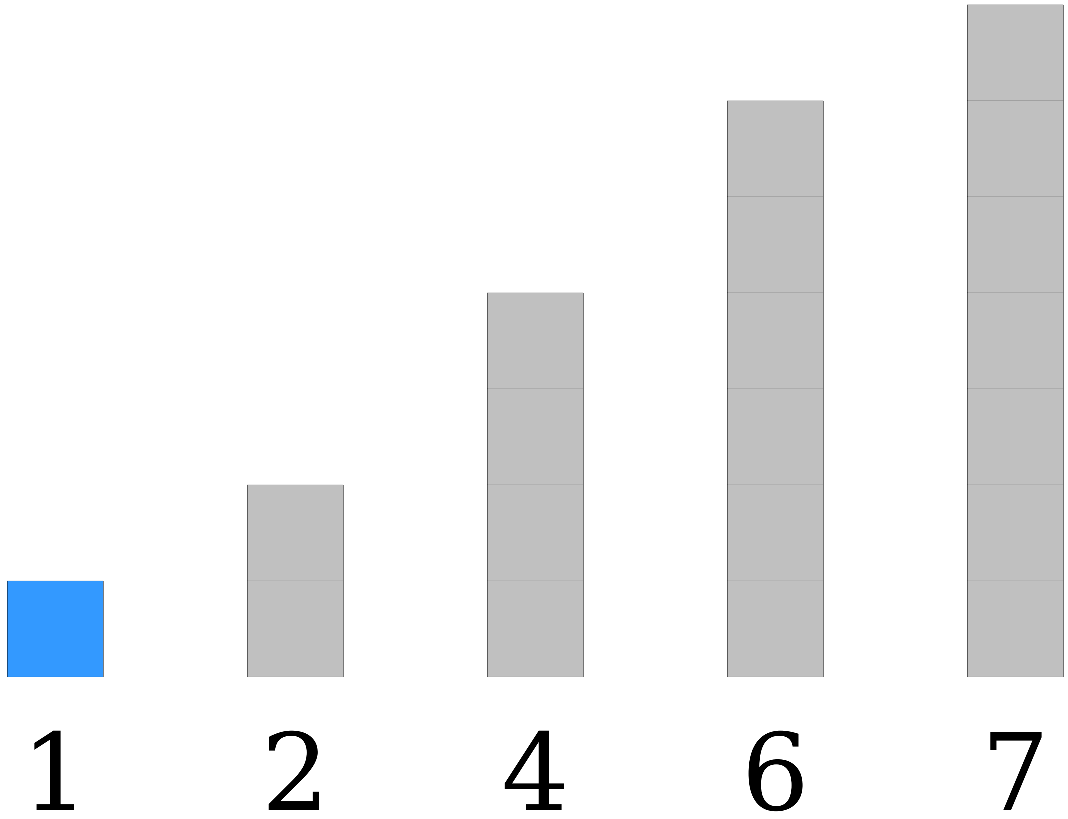
```
/**
 * Sorts the specified vector using insertion sort.
 *
 * @param v The vector to sort.
 */
void insertionSort(Vector<int>& v) {
  for (int i = 0; i < v.size(); i++) {
    /* Scan backwards until either (1) there is no
     * preceding element or the preceding element is
     * no bigger than us.
     */
    for (int j = i - 1; j >= 0; j--) {
      if (v[j] <= v[j + 1]) break;

      /* Swap this element back one step. */
      swap(v[j], v[j + 1]);
    }
  }
}
```

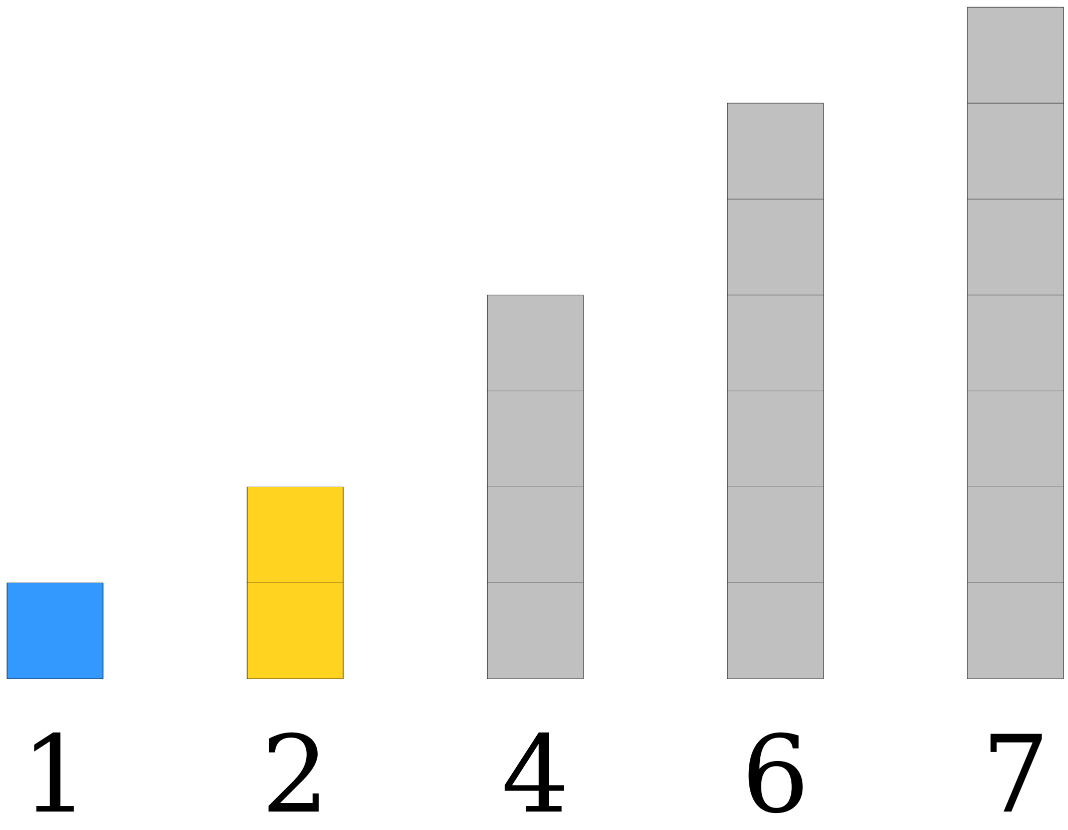
# How Fast is Insertion Sort?



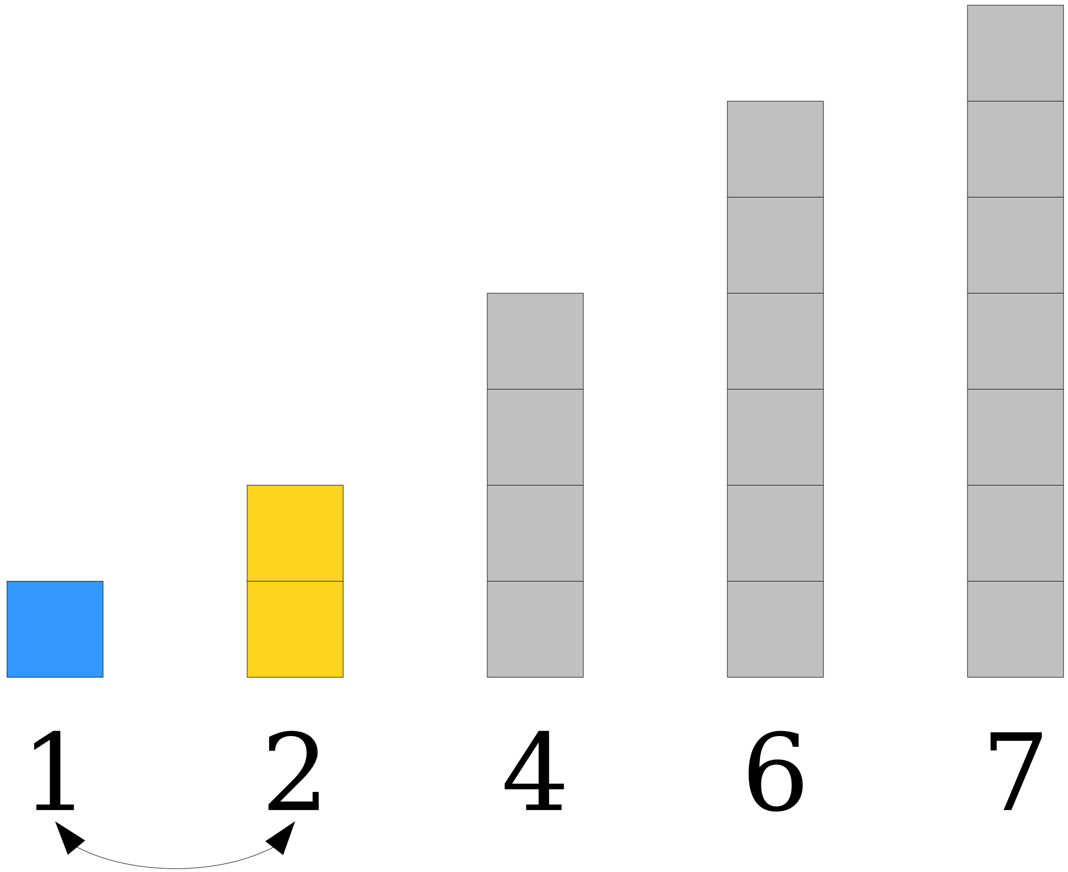
# How Fast is Insertion Sort?



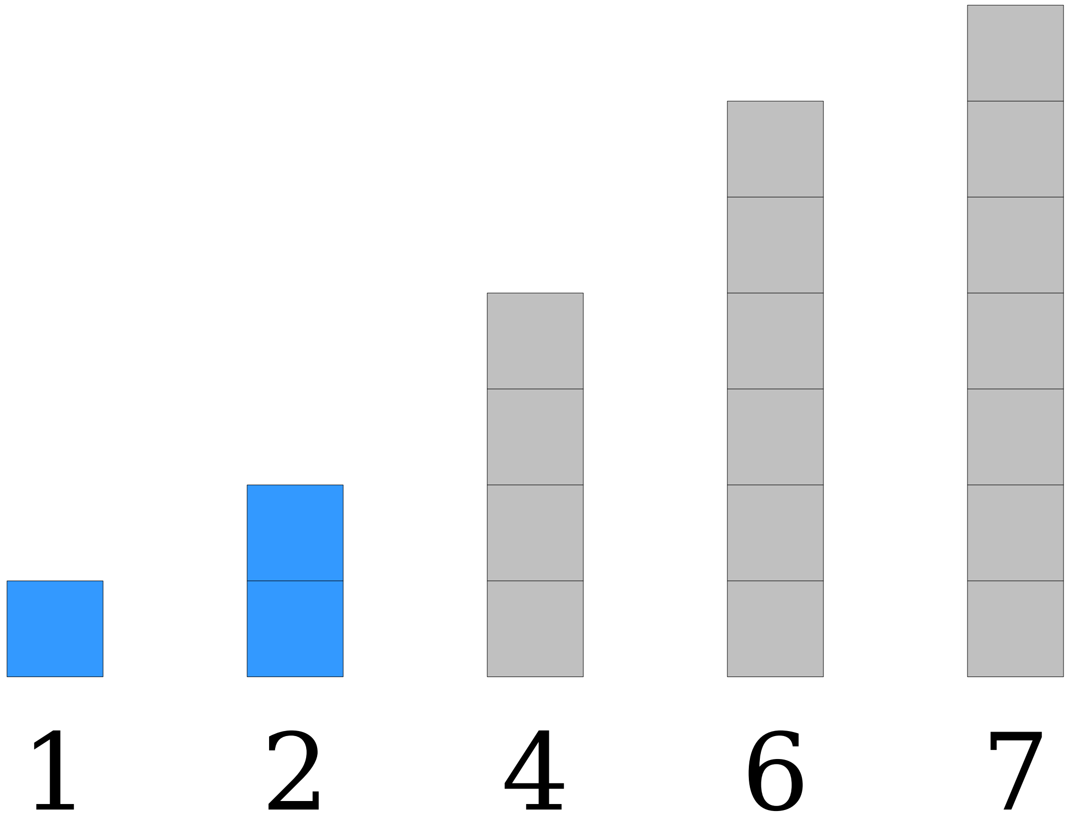
# How Fast is Insertion Sort?



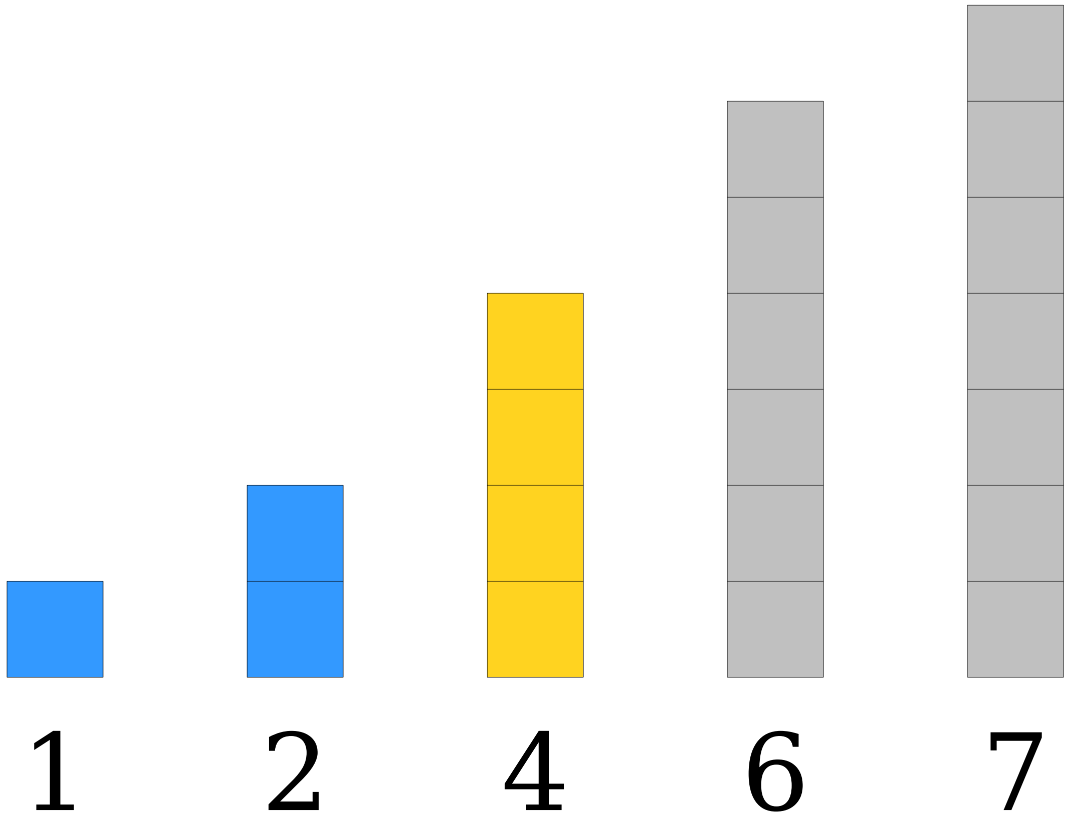
# How Fast is Insertion Sort?



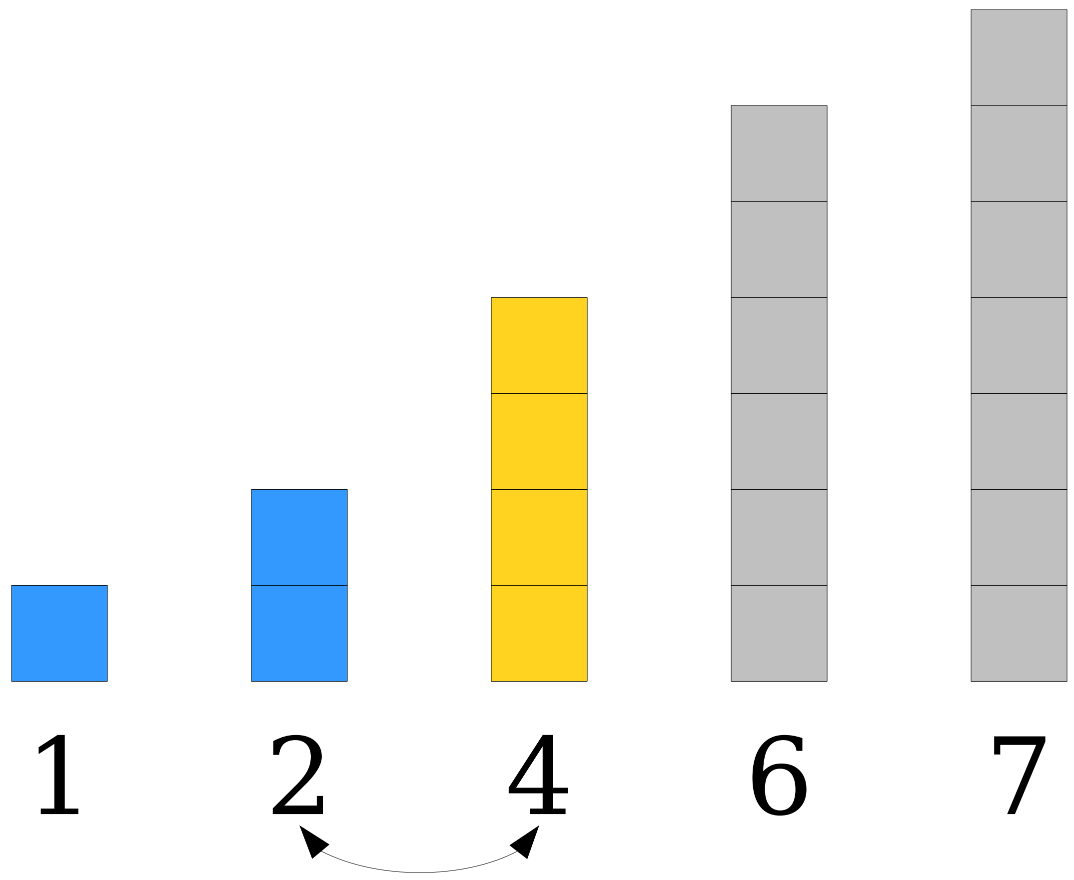
# How Fast is Insertion Sort?



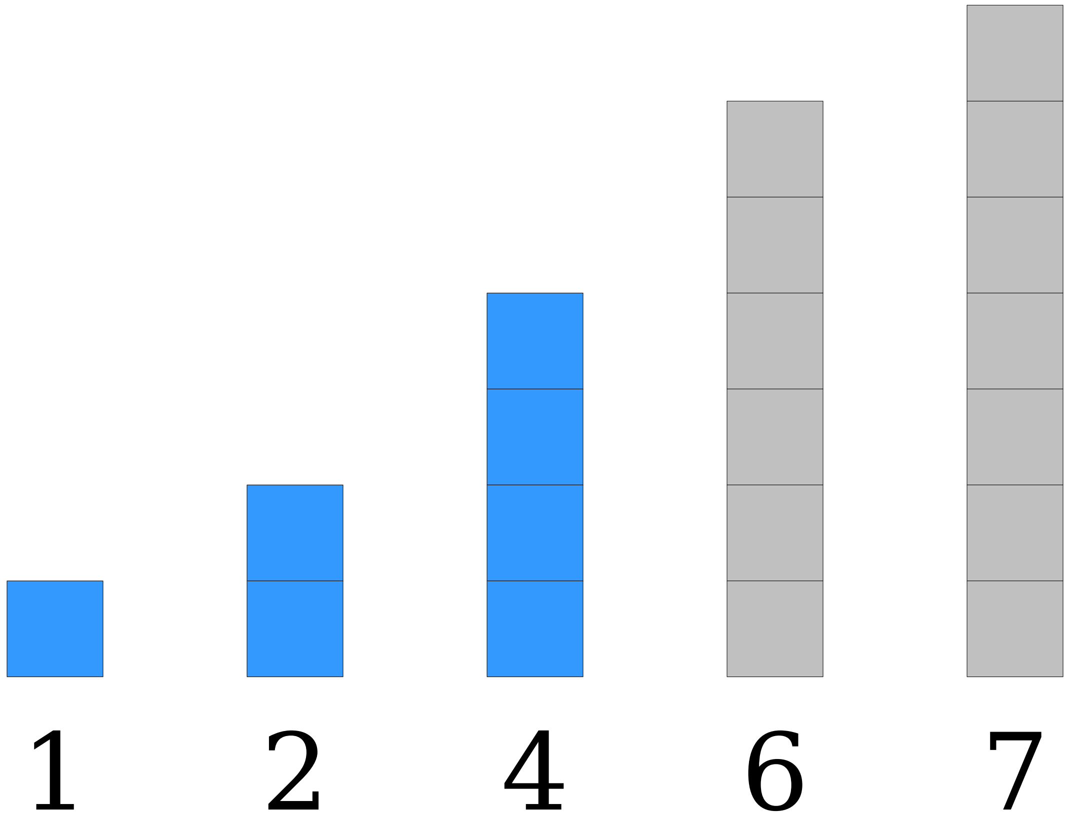
# How Fast is Insertion Sort?



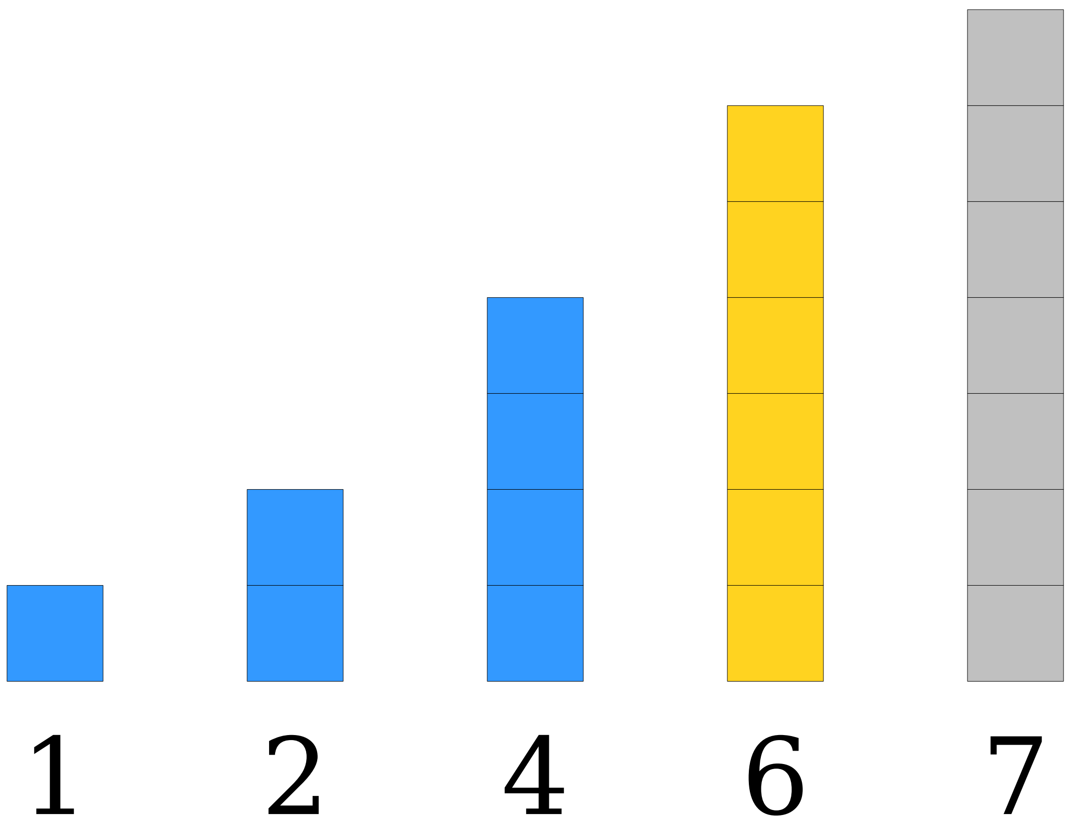
# How Fast is Insertion Sort?



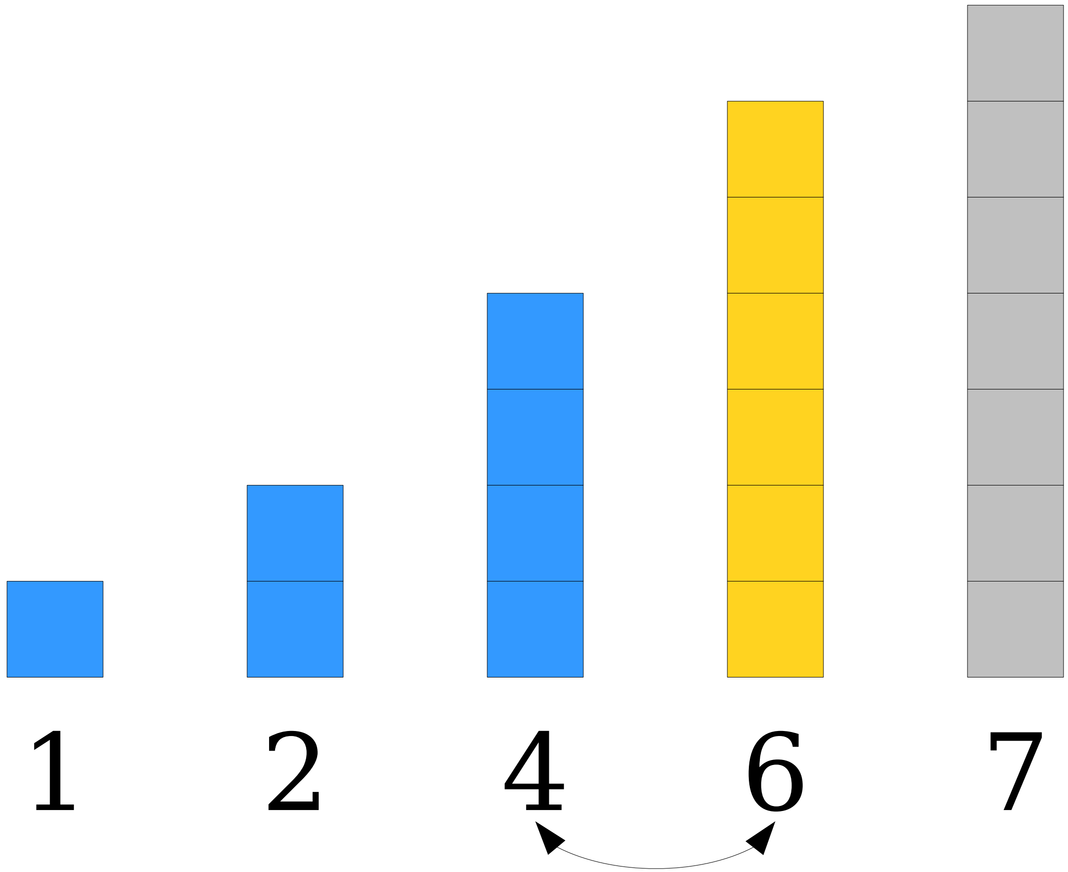
# How Fast is Insertion Sort?



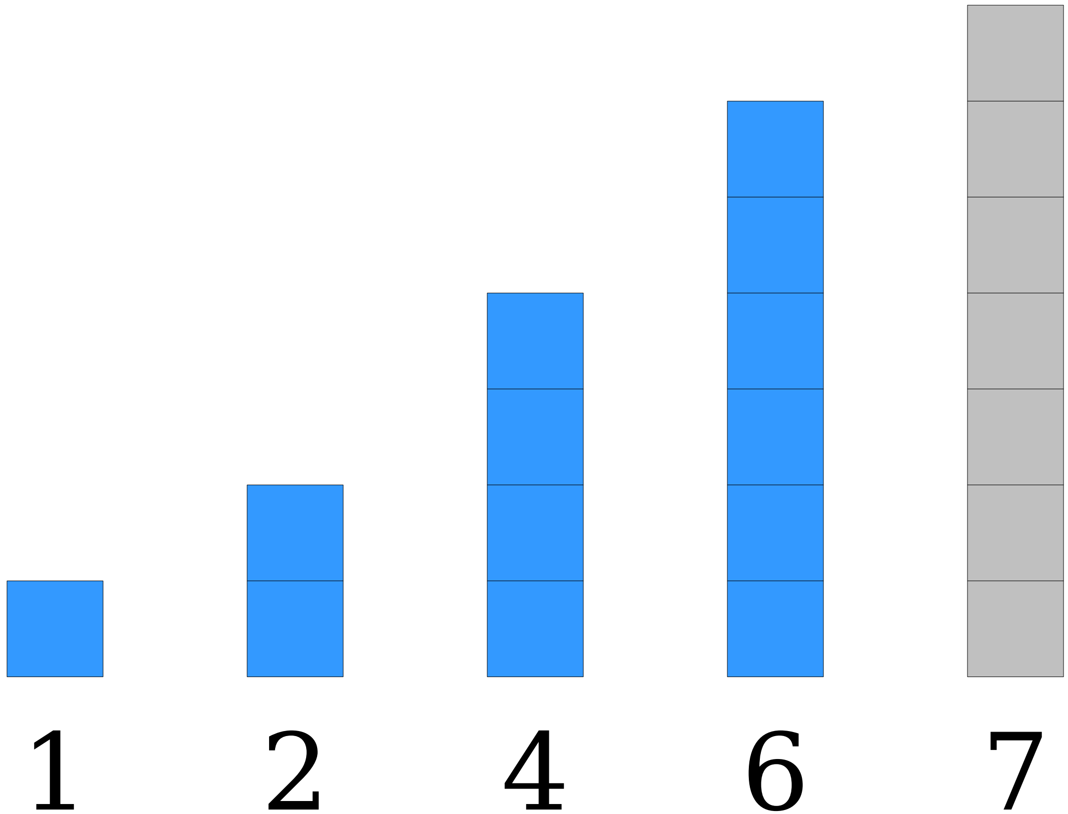
# How Fast is Insertion Sort?



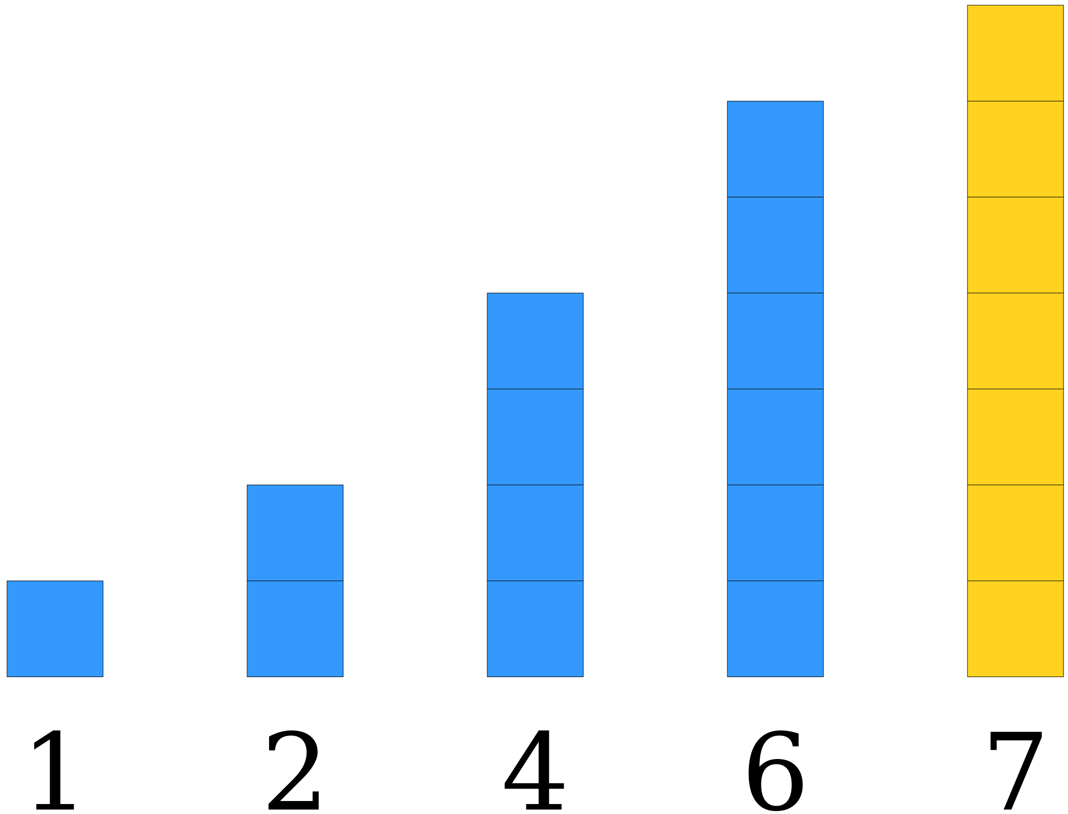
# How Fast is Insertion Sort?



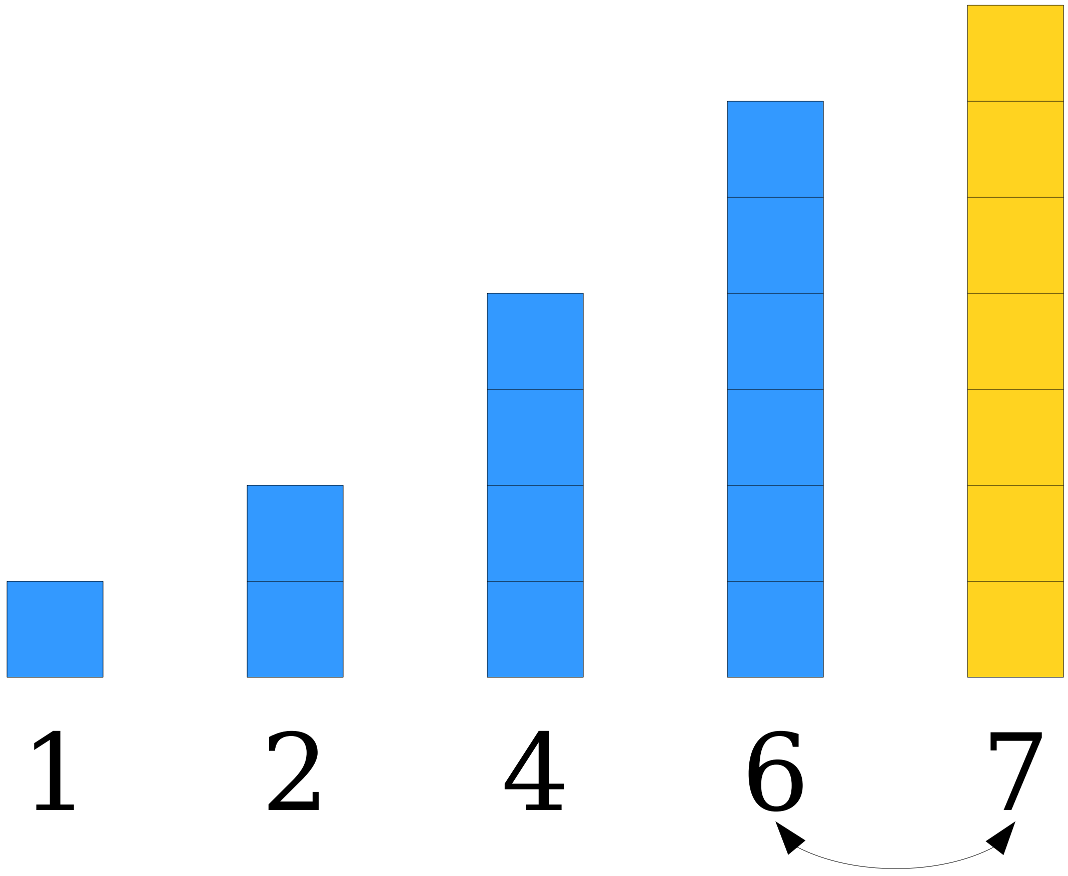
# How Fast is Insertion Sort?



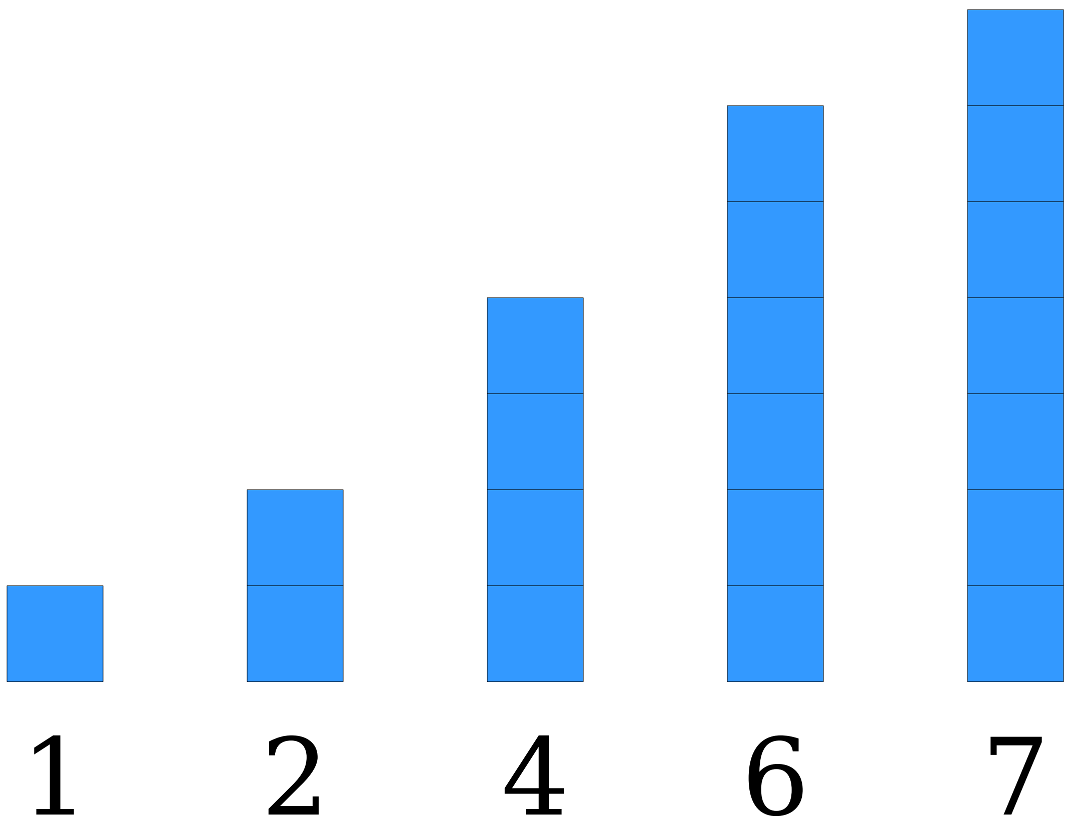
# How Fast is Insertion Sort?



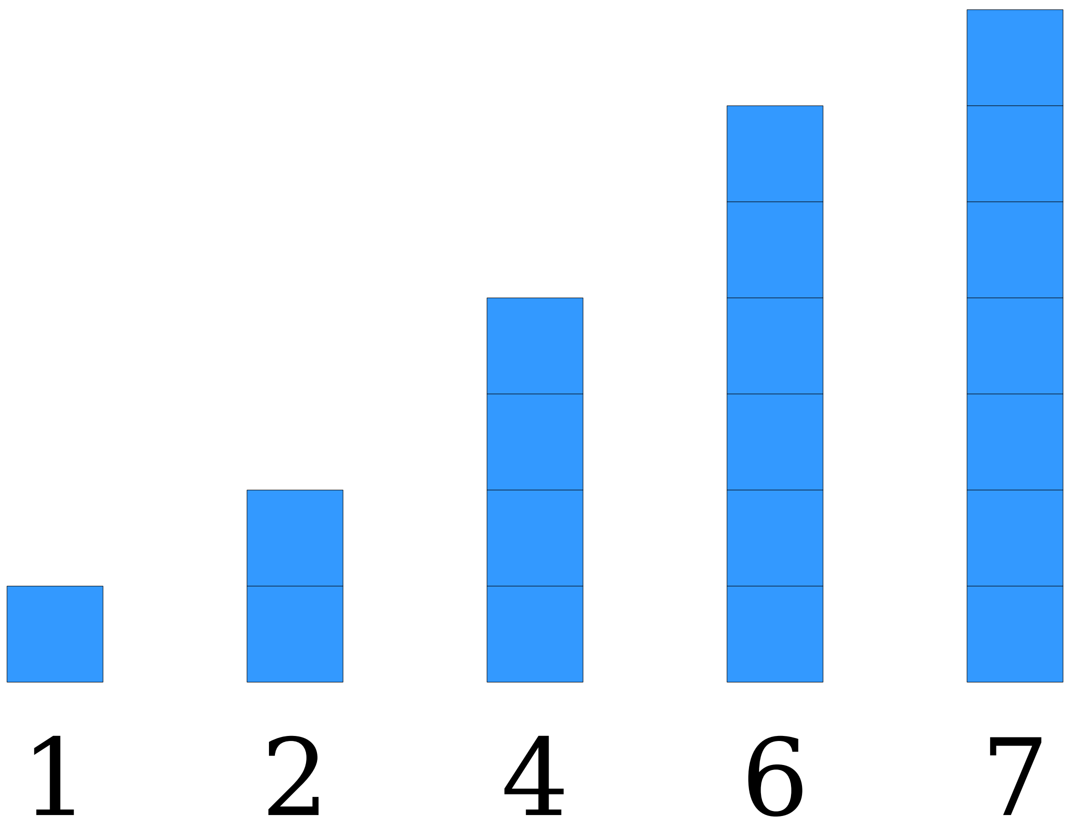
# How Fast is Insertion Sort?



# How Fast is Insertion Sort?

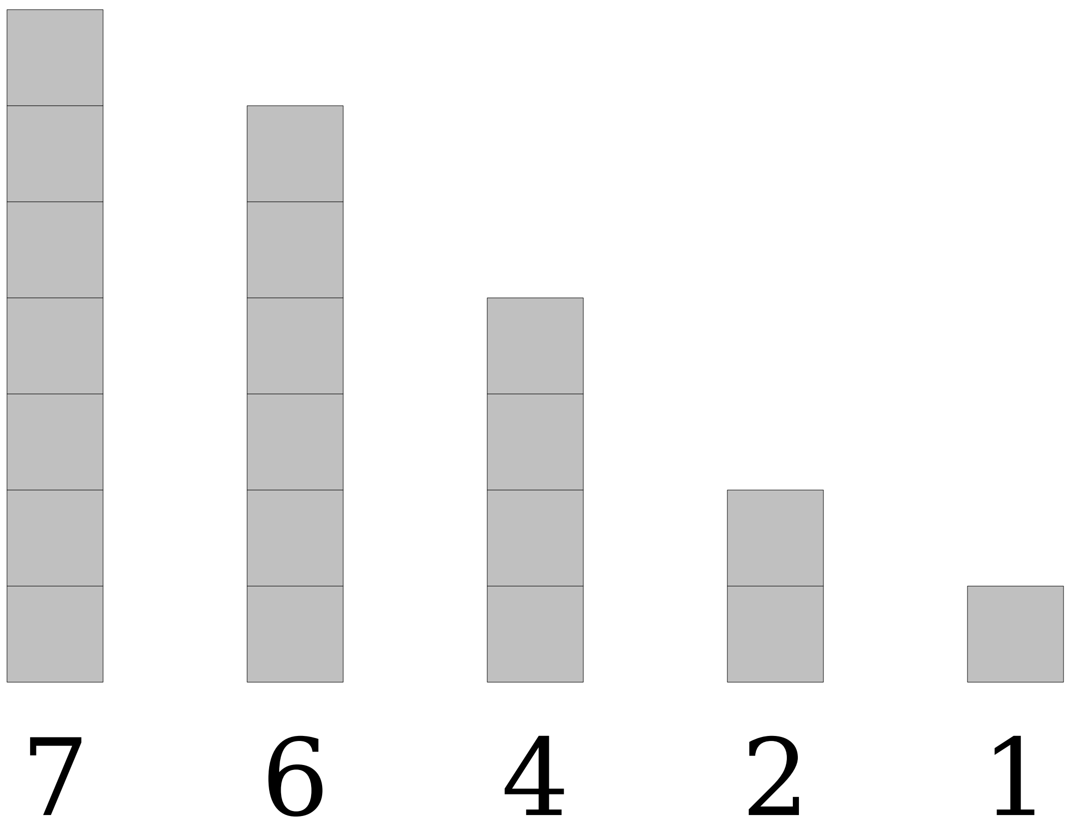


# How Fast is Insertion Sort?

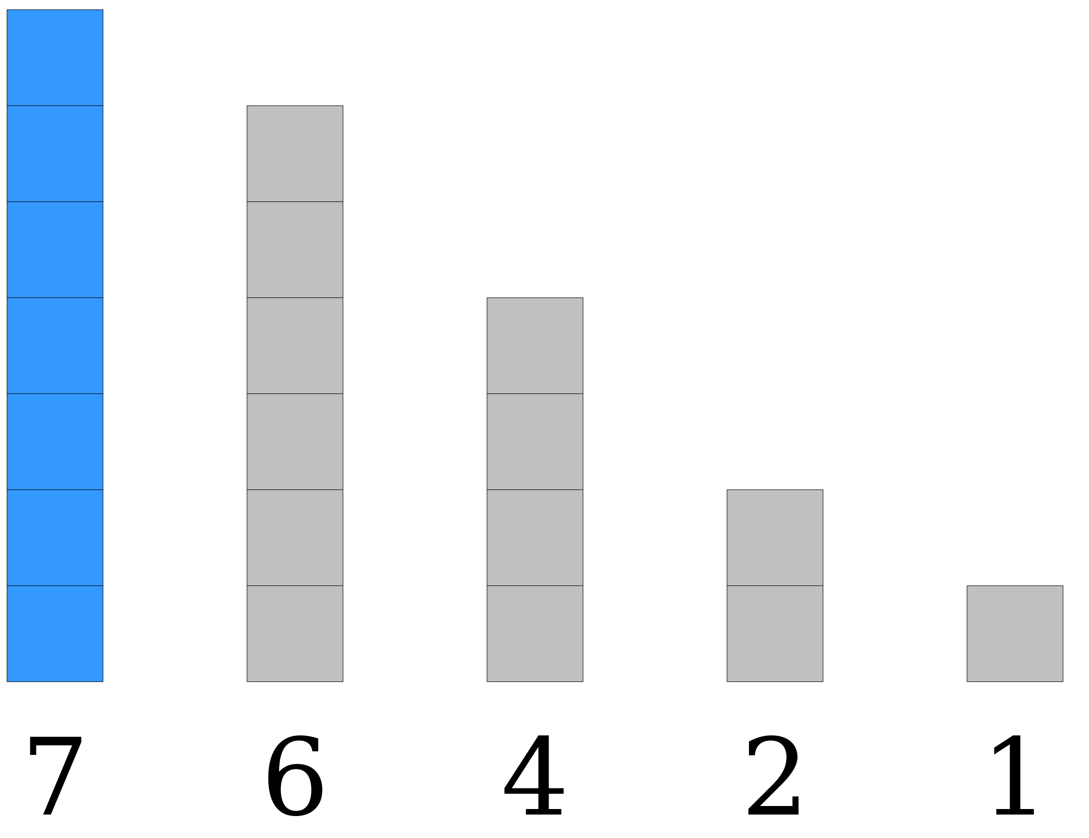



Work done: $\mathbf{O(\boldsymbol{n})}$

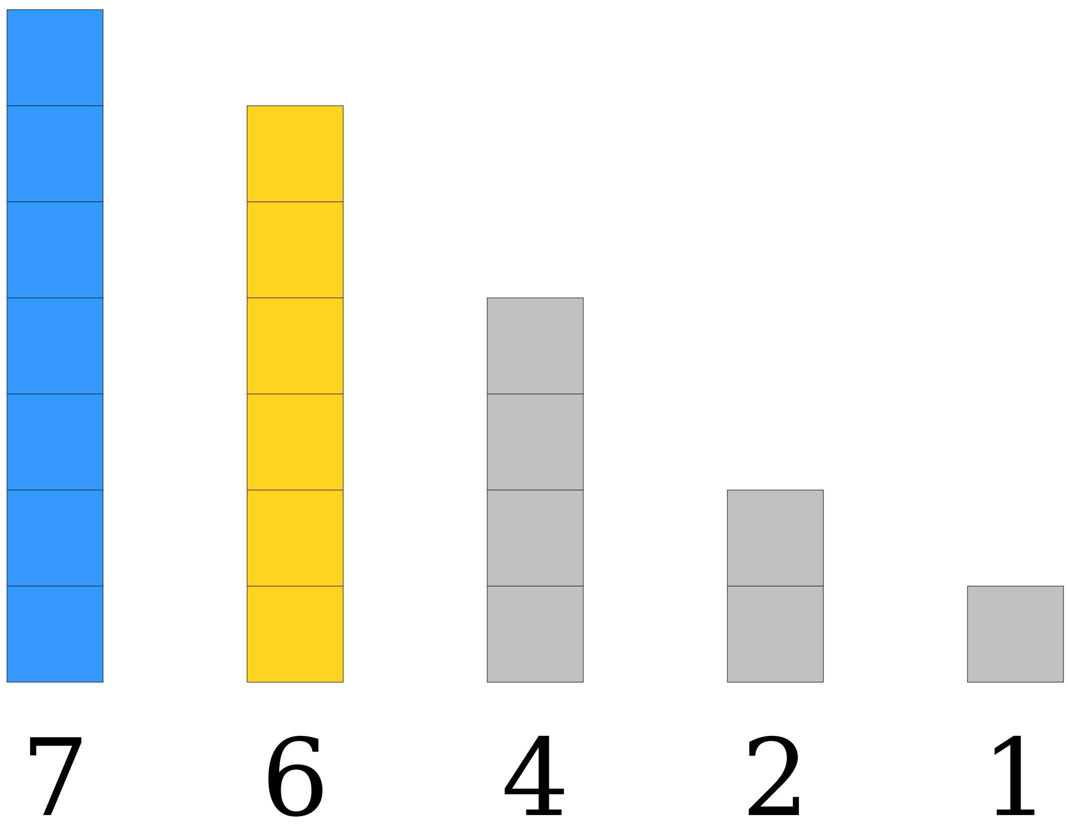
# How Fast is Insertion Sort?



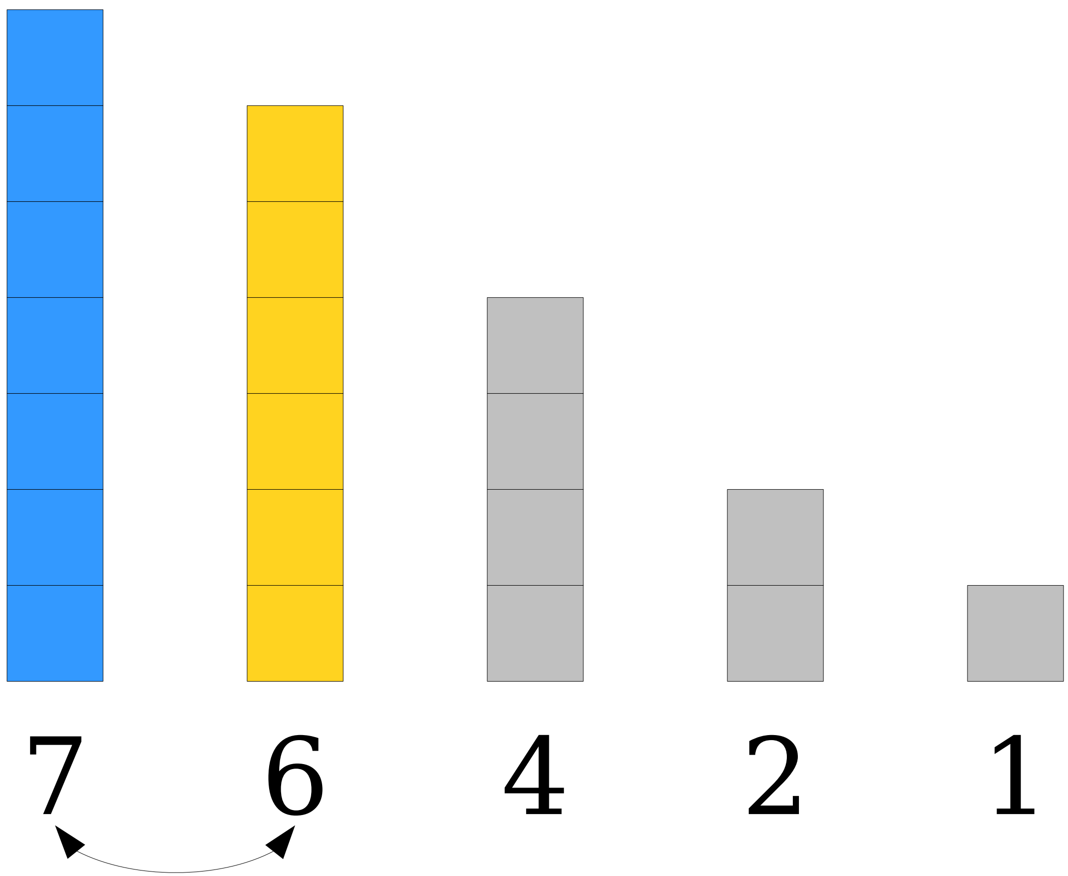
# How Fast is Insertion Sort?



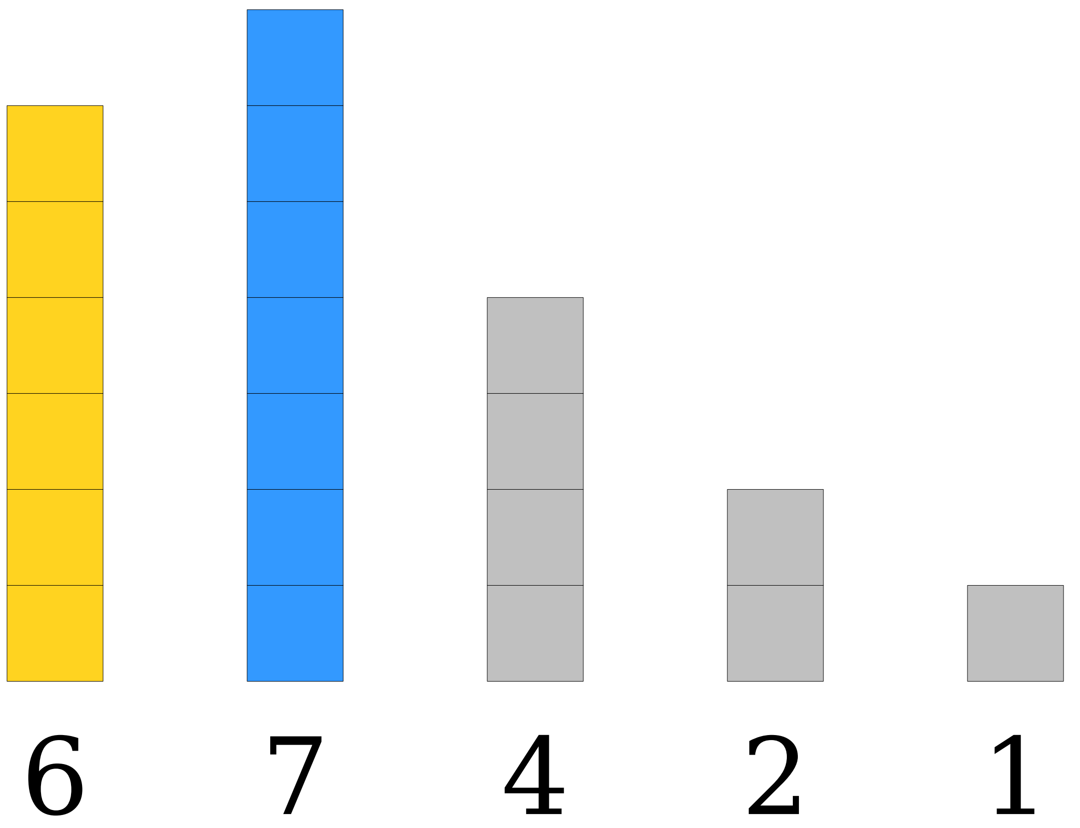
# How Fast is Insertion Sort?



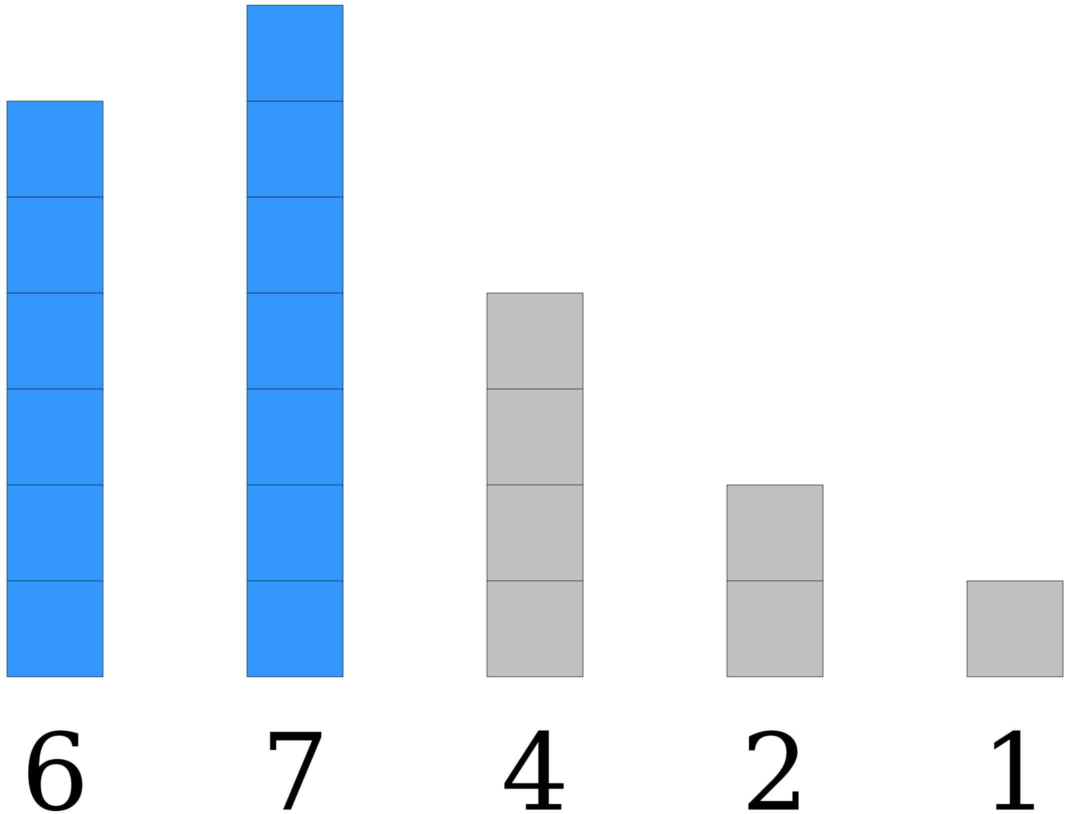
# How Fast is Insertion Sort?



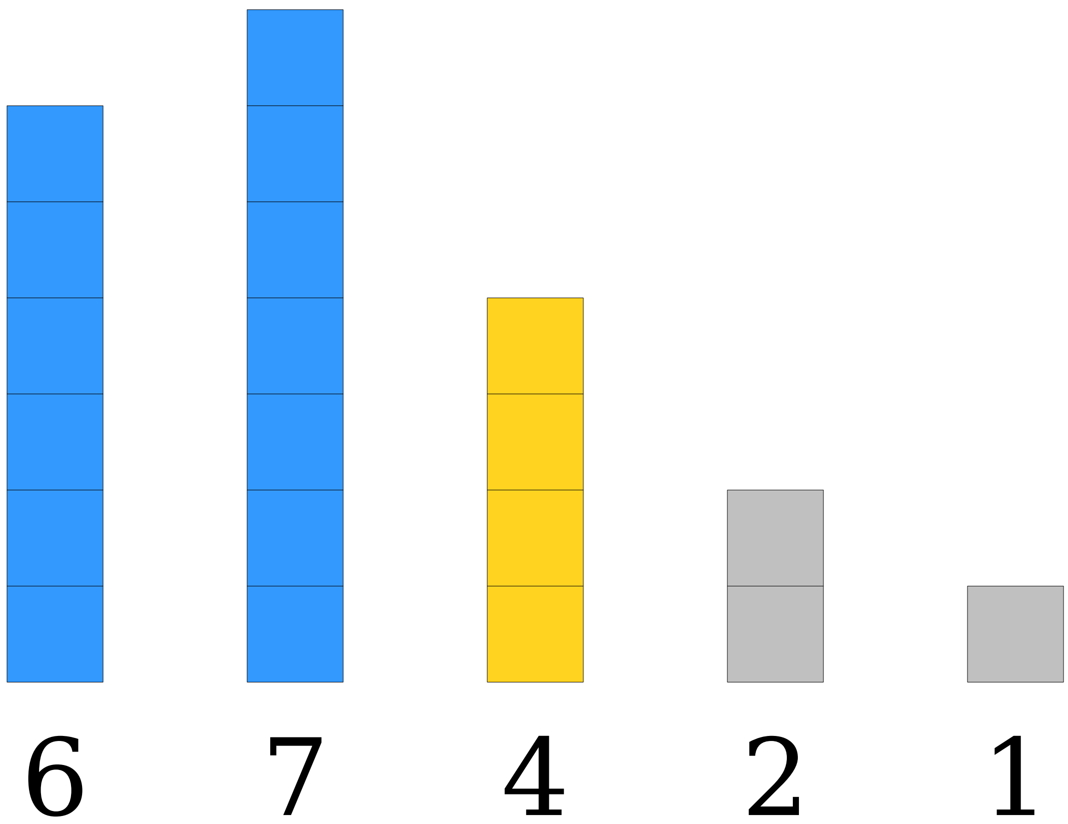
# How Fast is Insertion Sort?



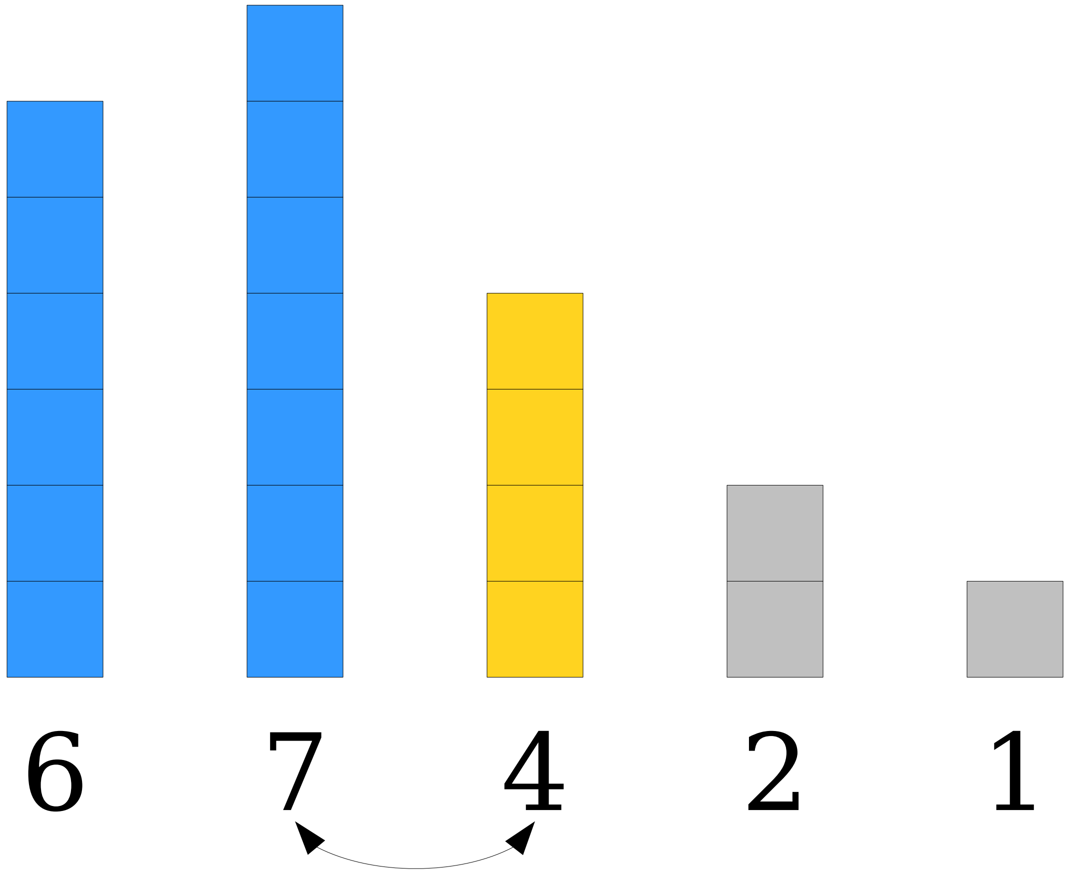
# How Fast is Insertion Sort?



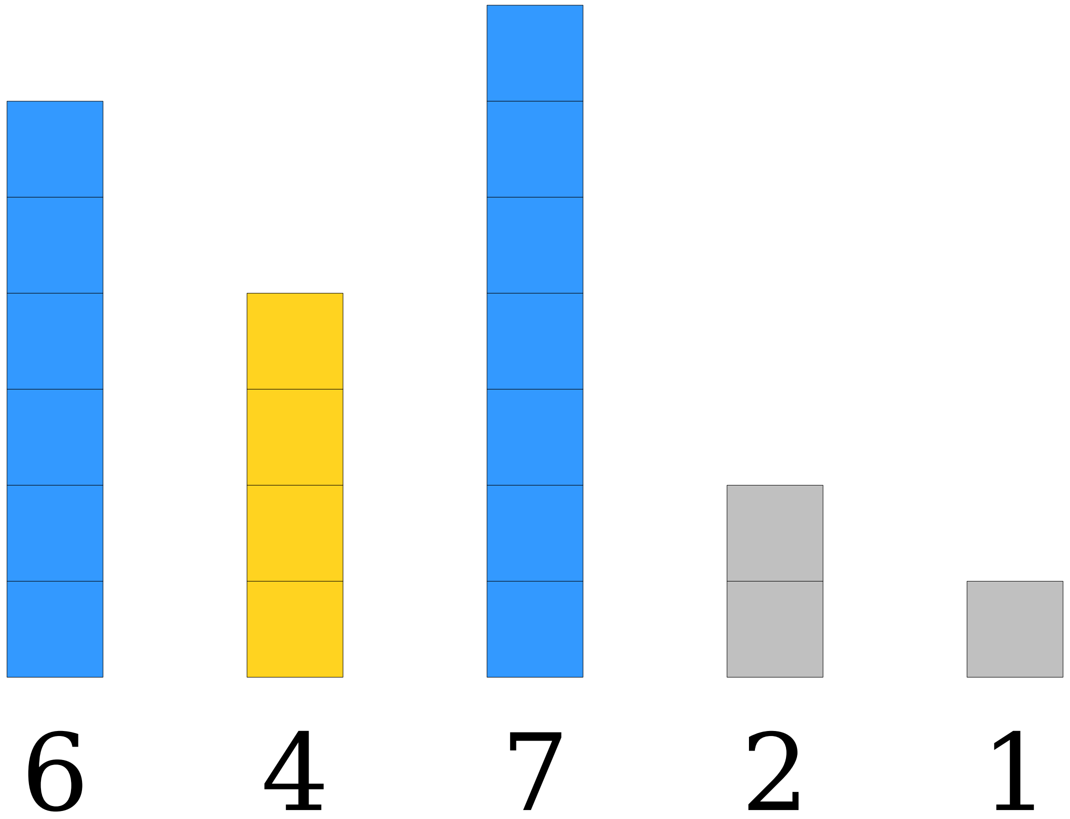
# How Fast is Insertion Sort?



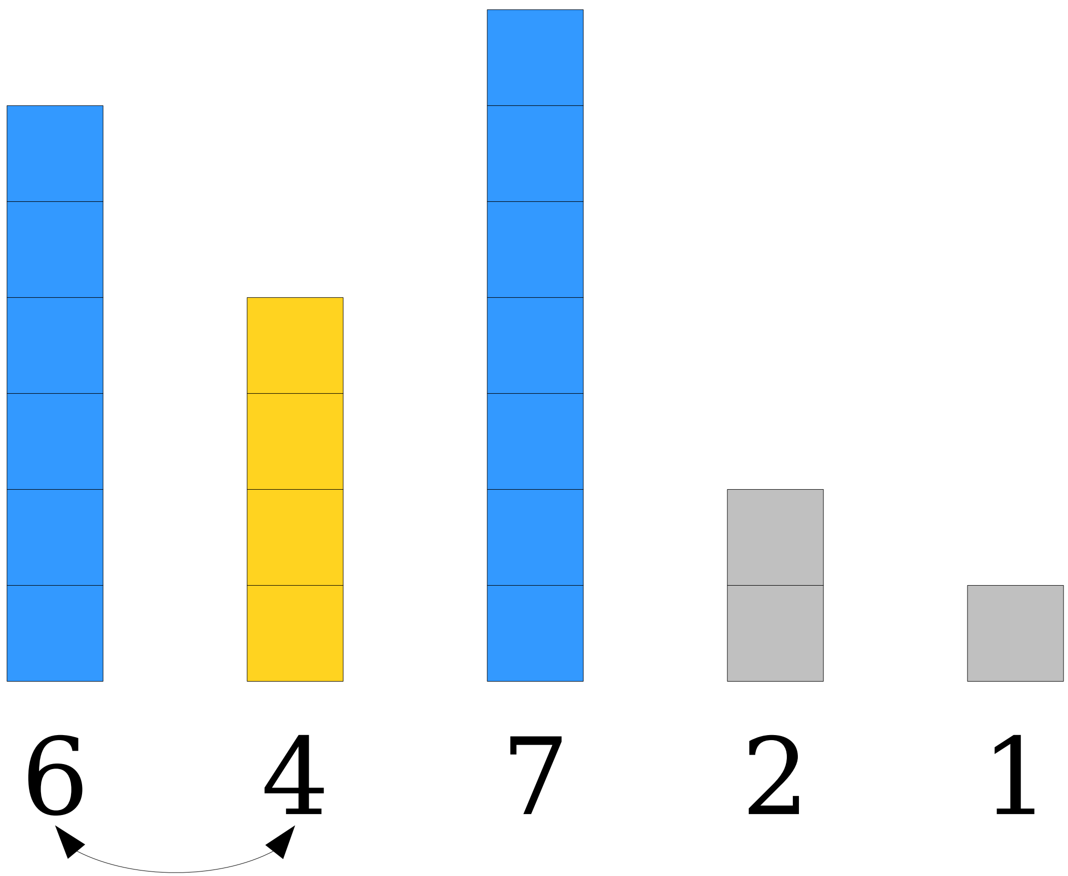
# How Fast is Insertion Sort?



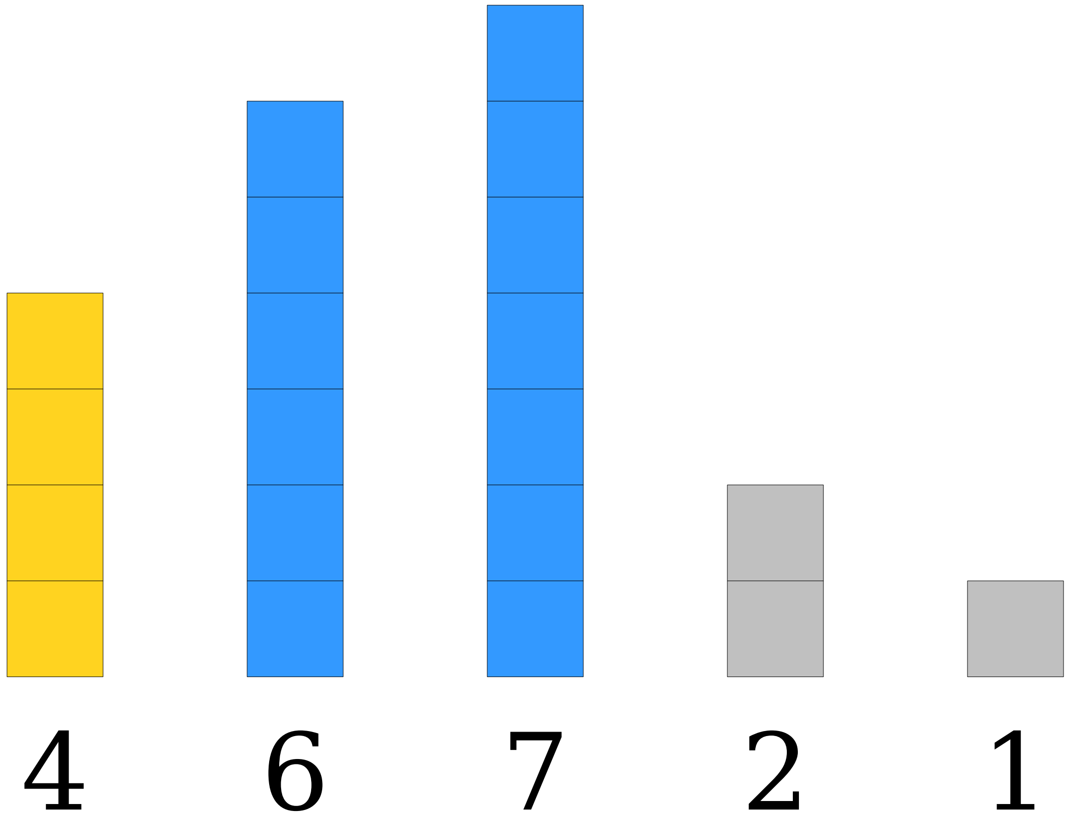
# How Fast is Insertion Sort?



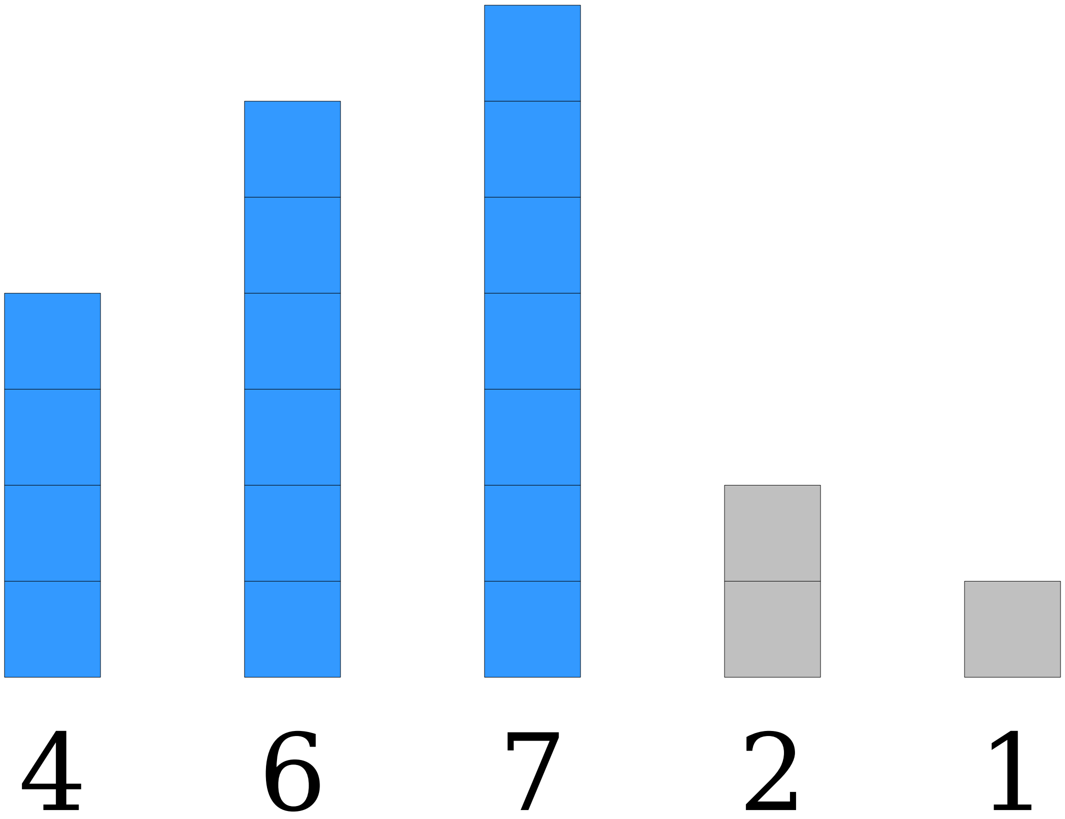
# How Fast is Insertion Sort?



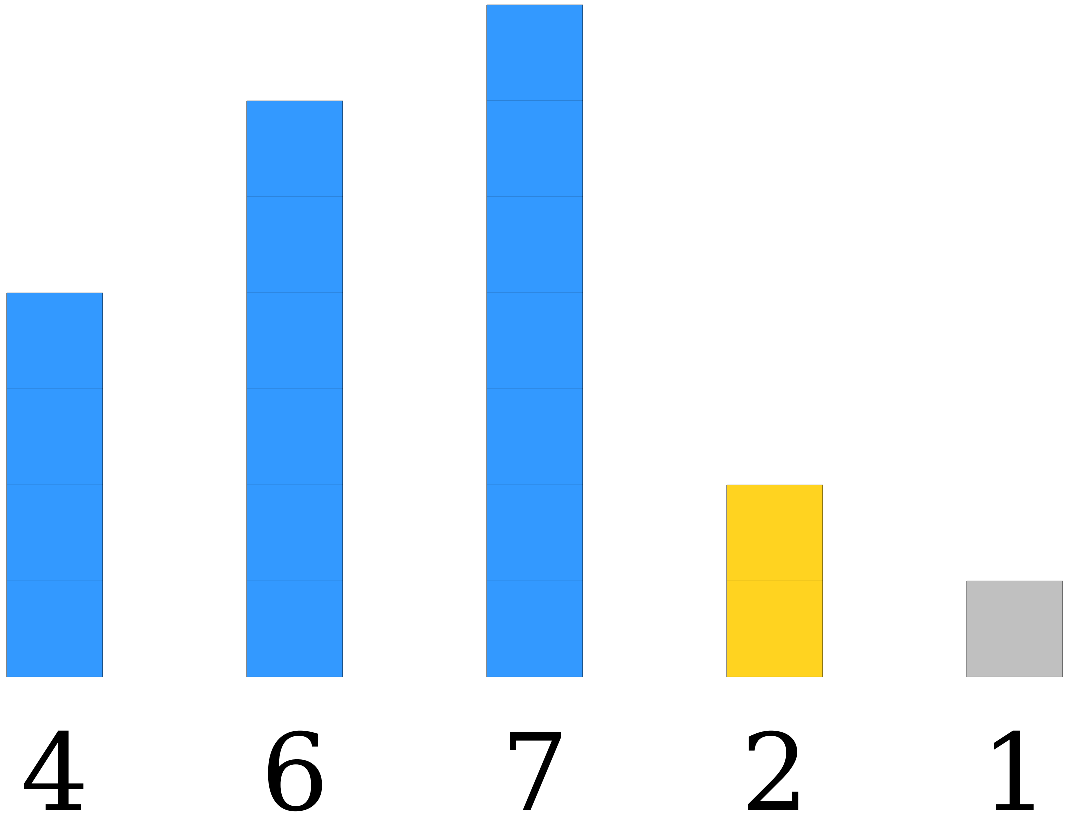
# How Fast is Insertion Sort?



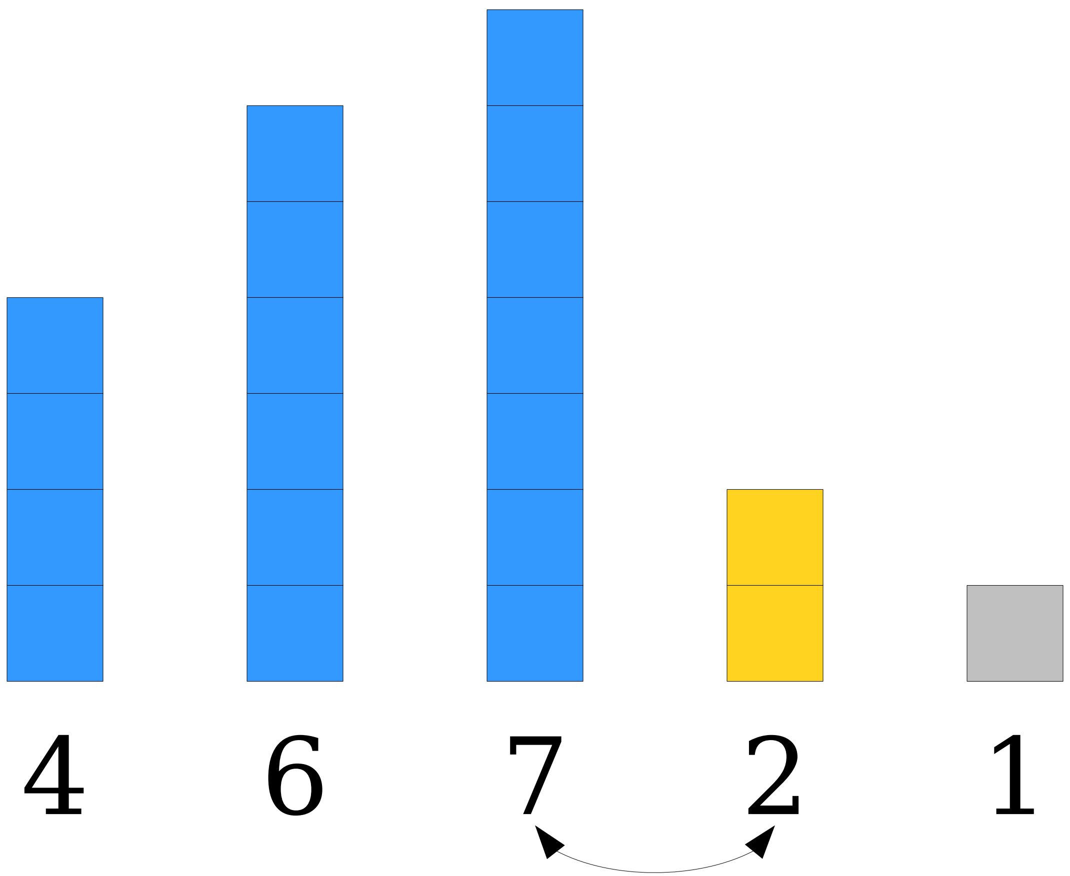
# How Fast is Insertion Sort?



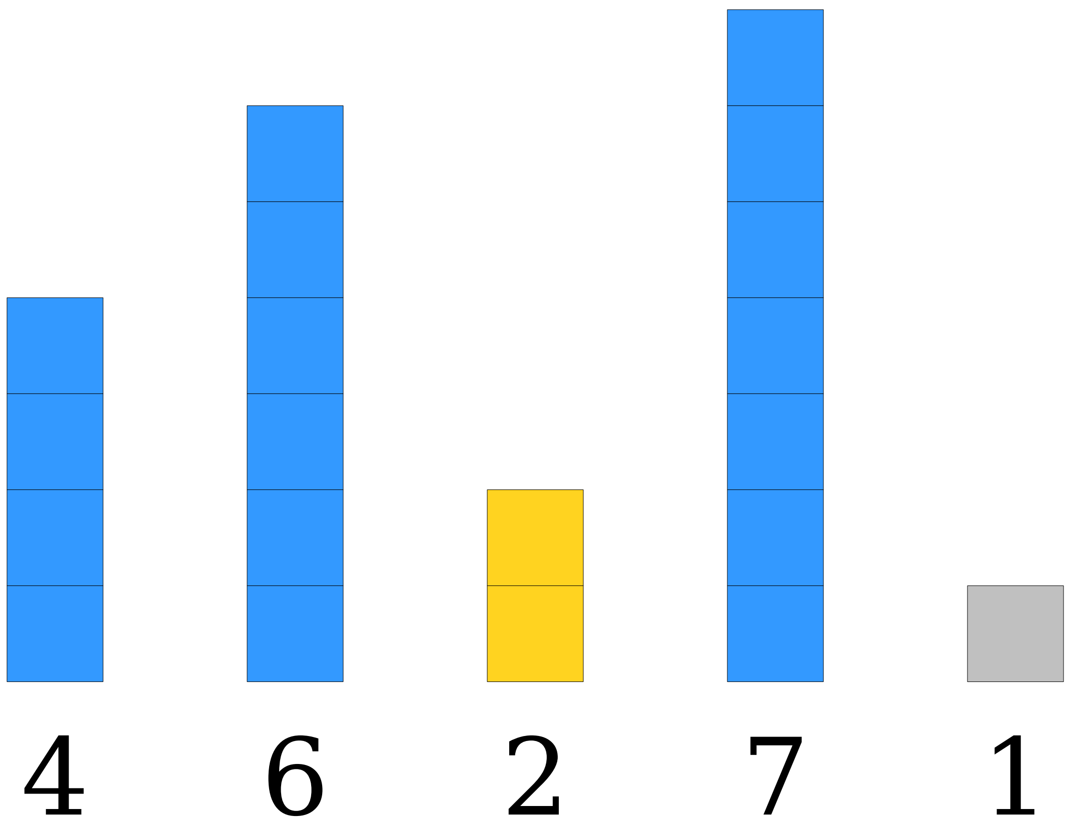
# How Fast is Insertion Sort?



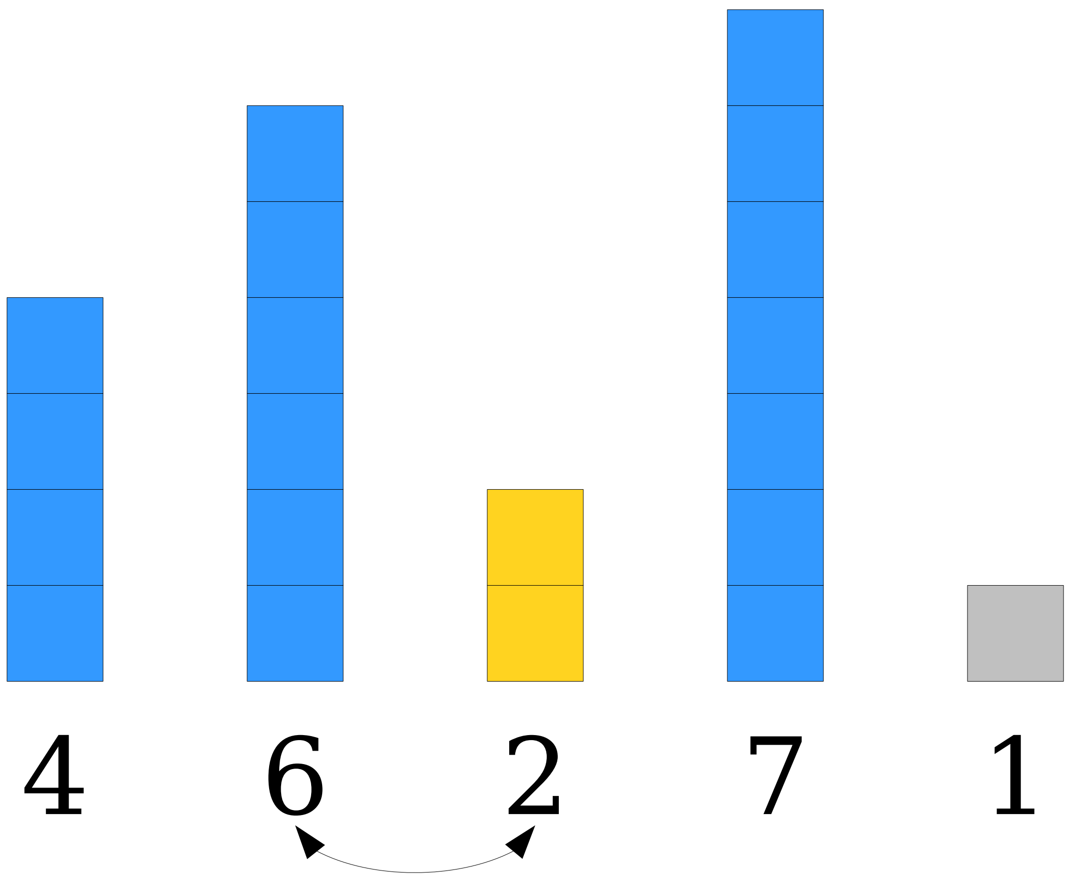
# How Fast is Insertion Sort?



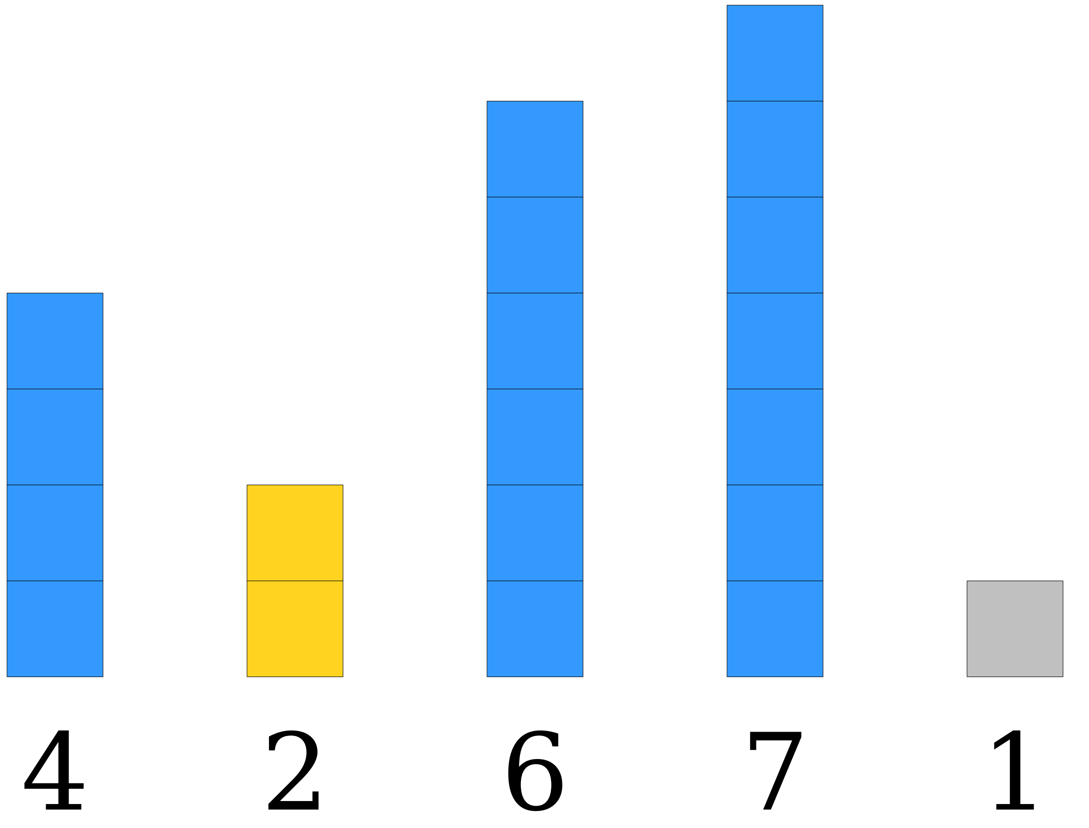
# How Fast is Insertion Sort?



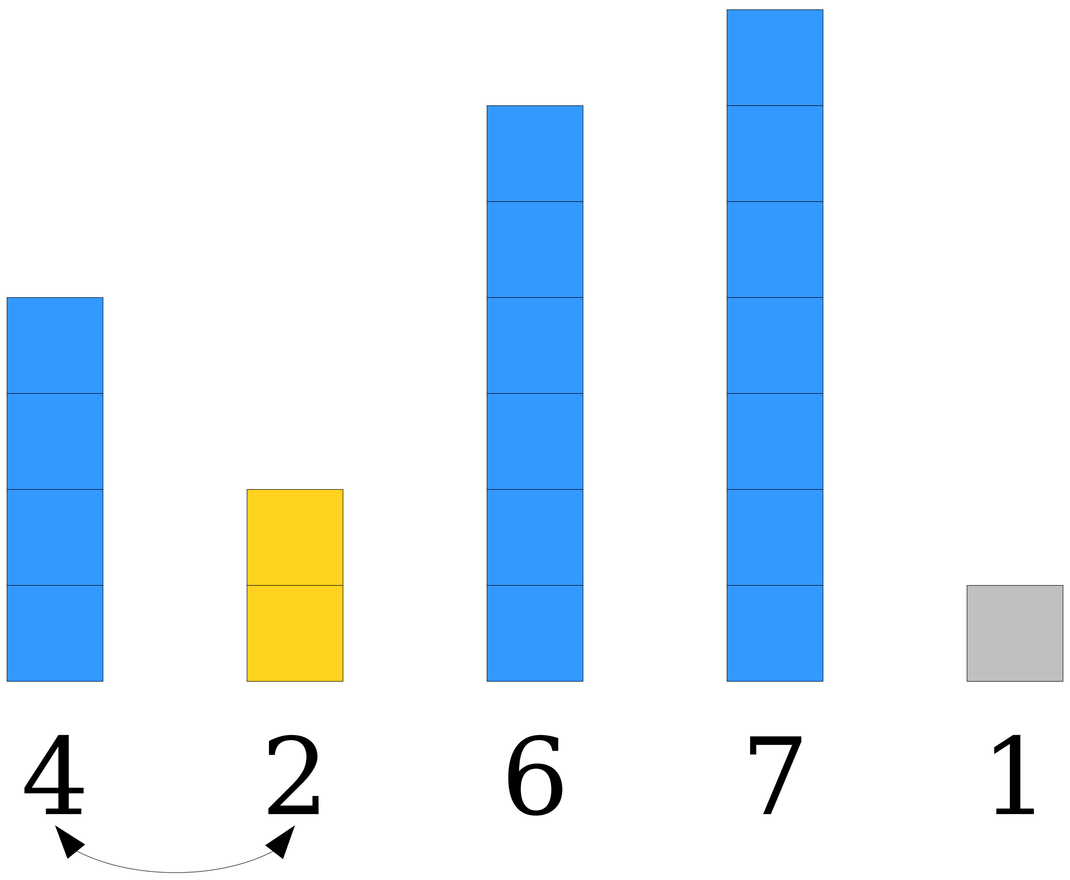
# How Fast is Insertion Sort?



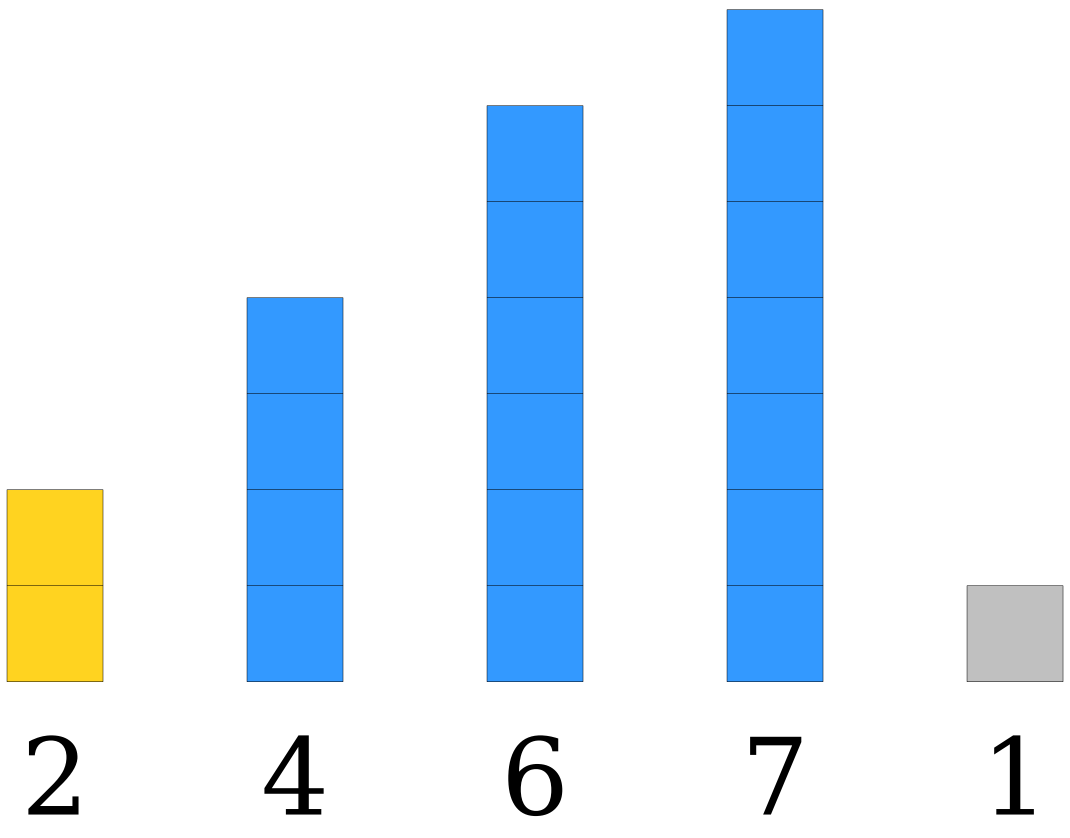
# How Fast is Insertion Sort?



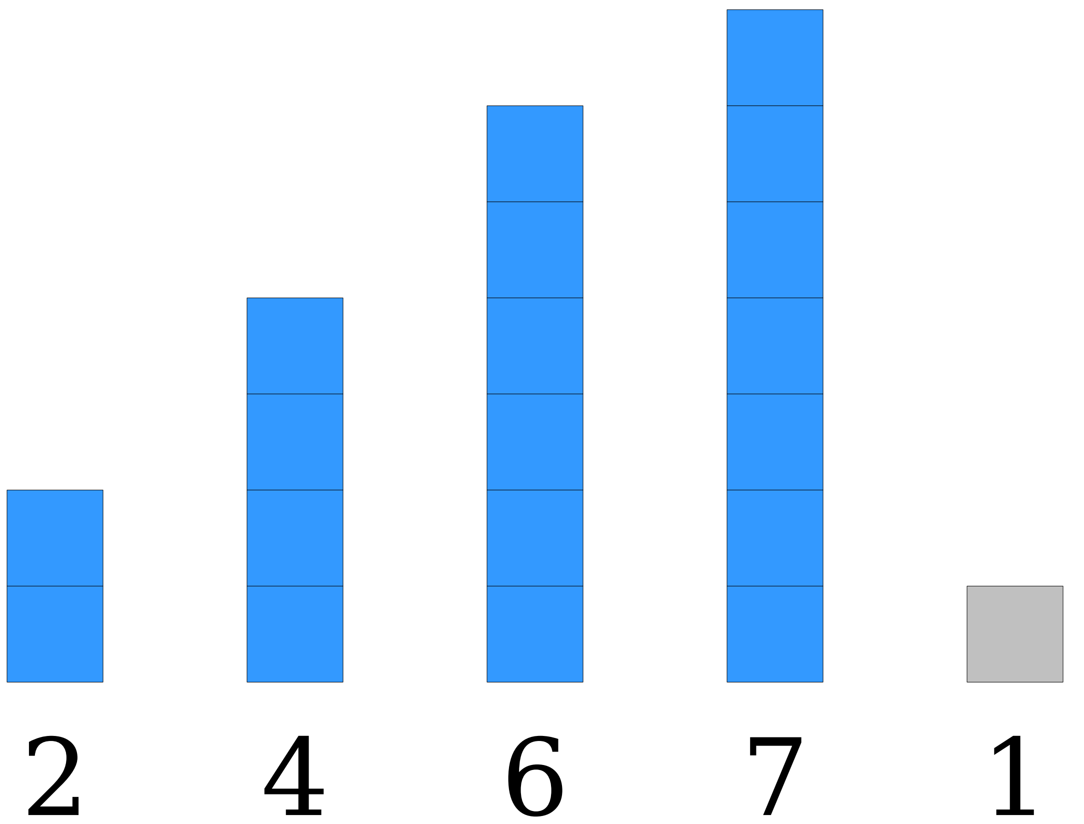
# How Fast is Insertion Sort?



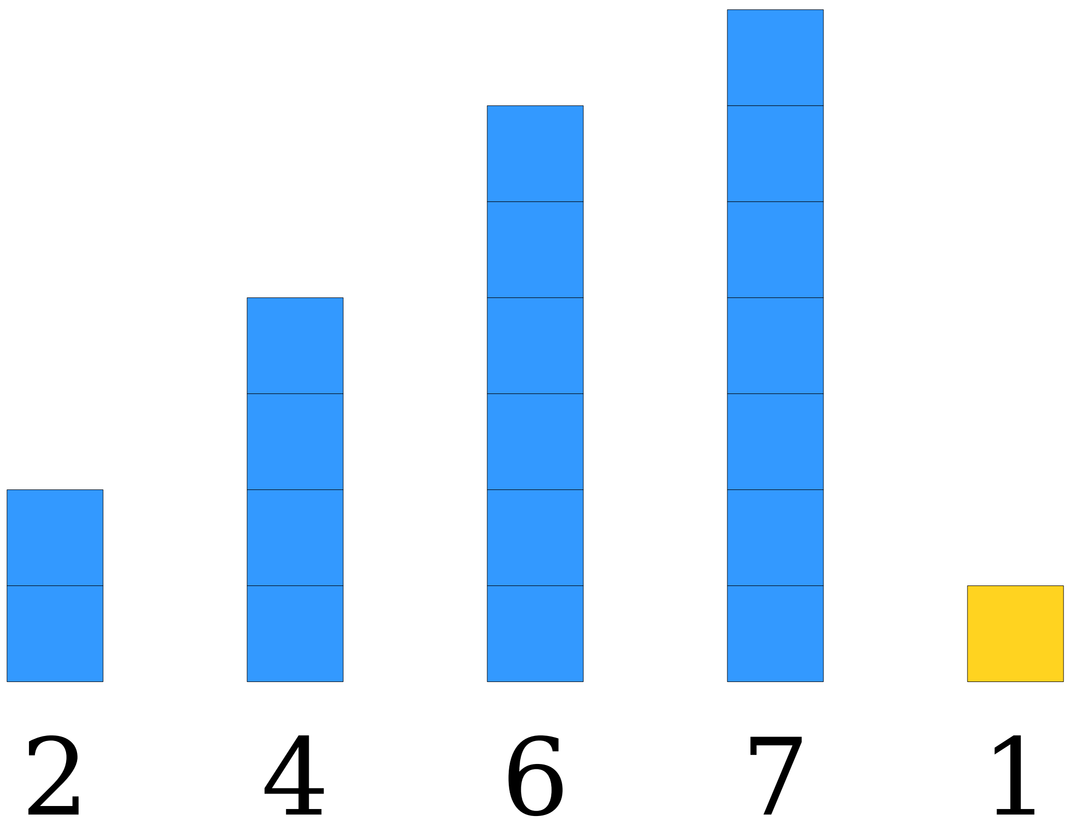
# How Fast is Insertion Sort?



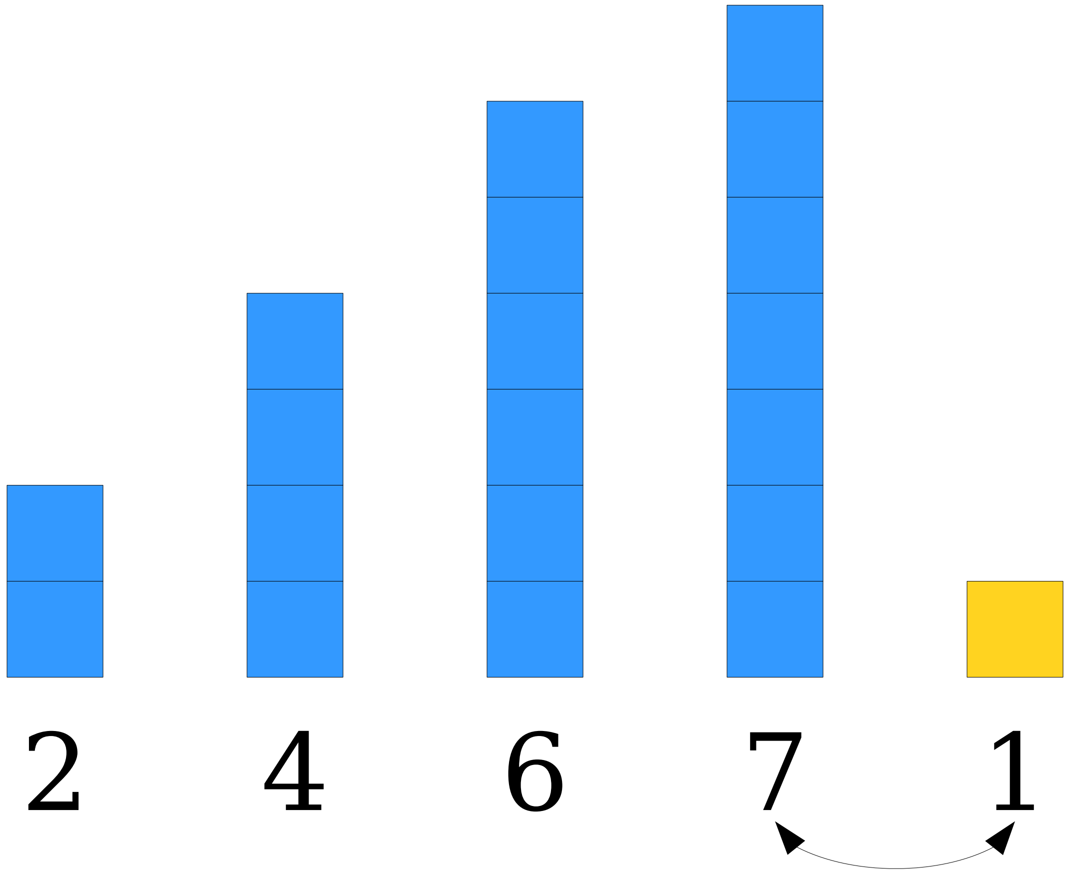
# How Fast is Insertion Sort?



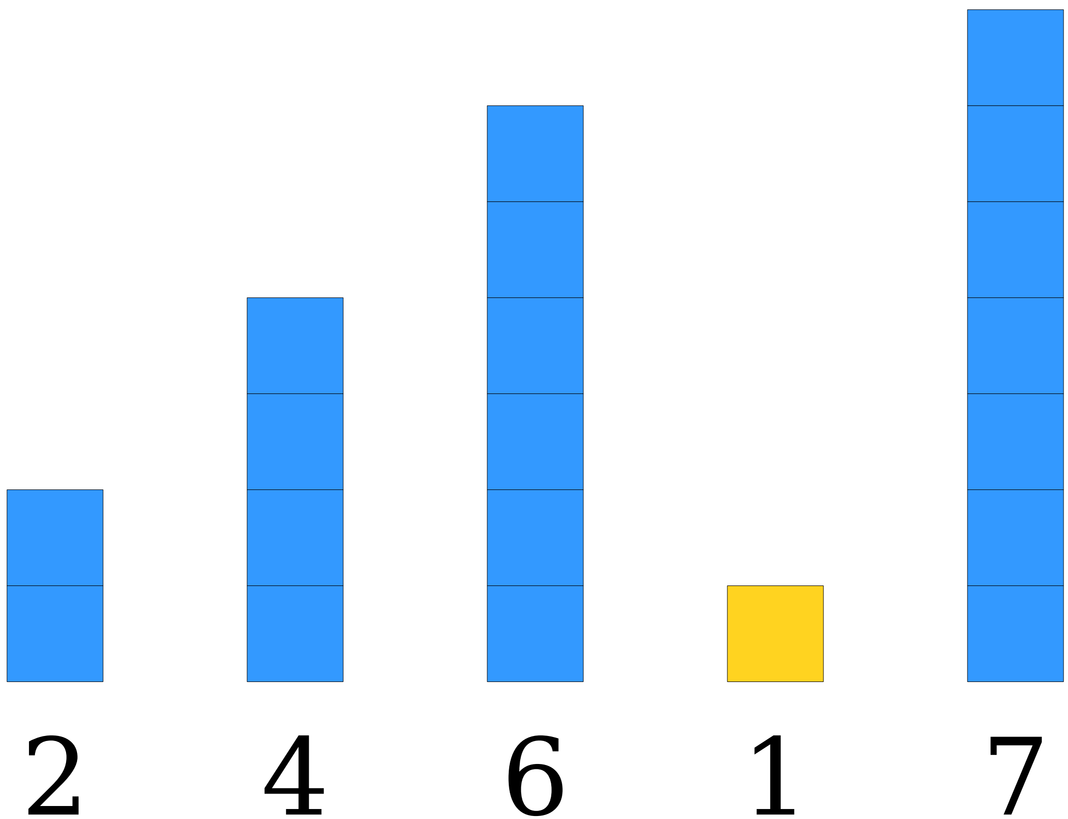
# How Fast is Insertion Sort?



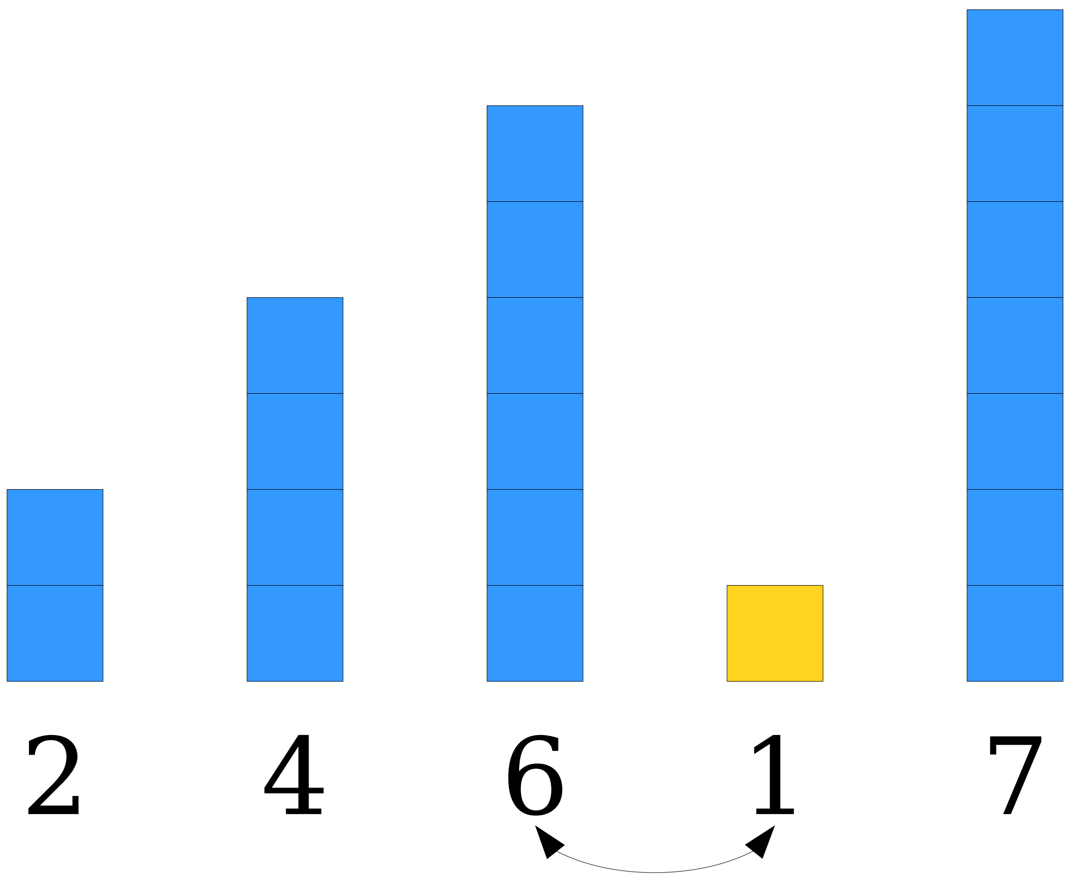
# How Fast is Insertion Sort?



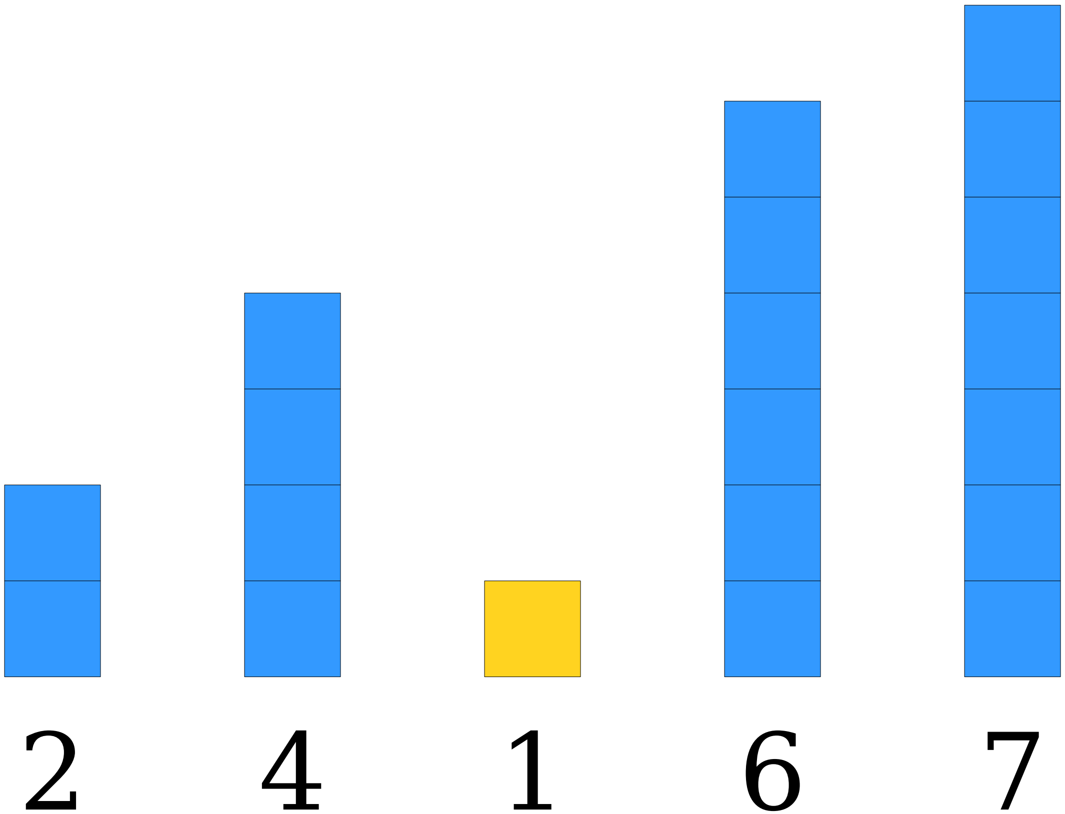
# How Fast is Insertion Sort?



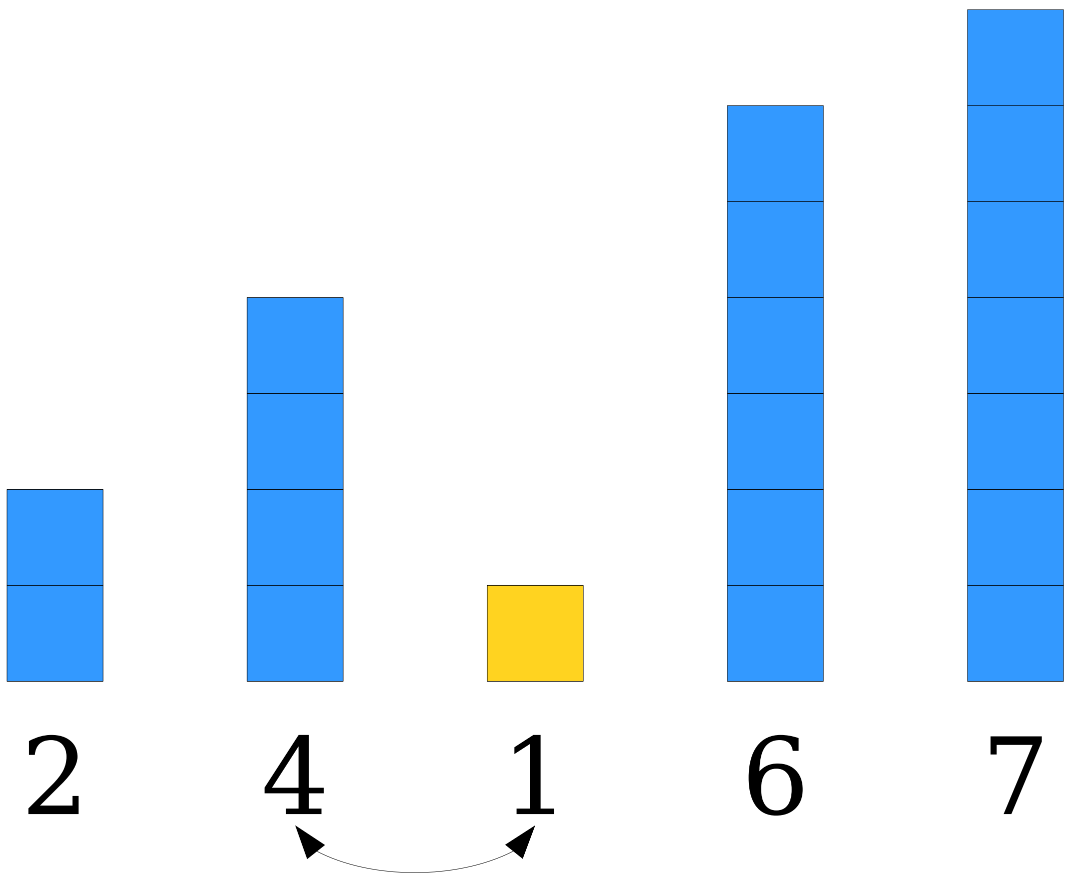
# How Fast is Insertion Sort?



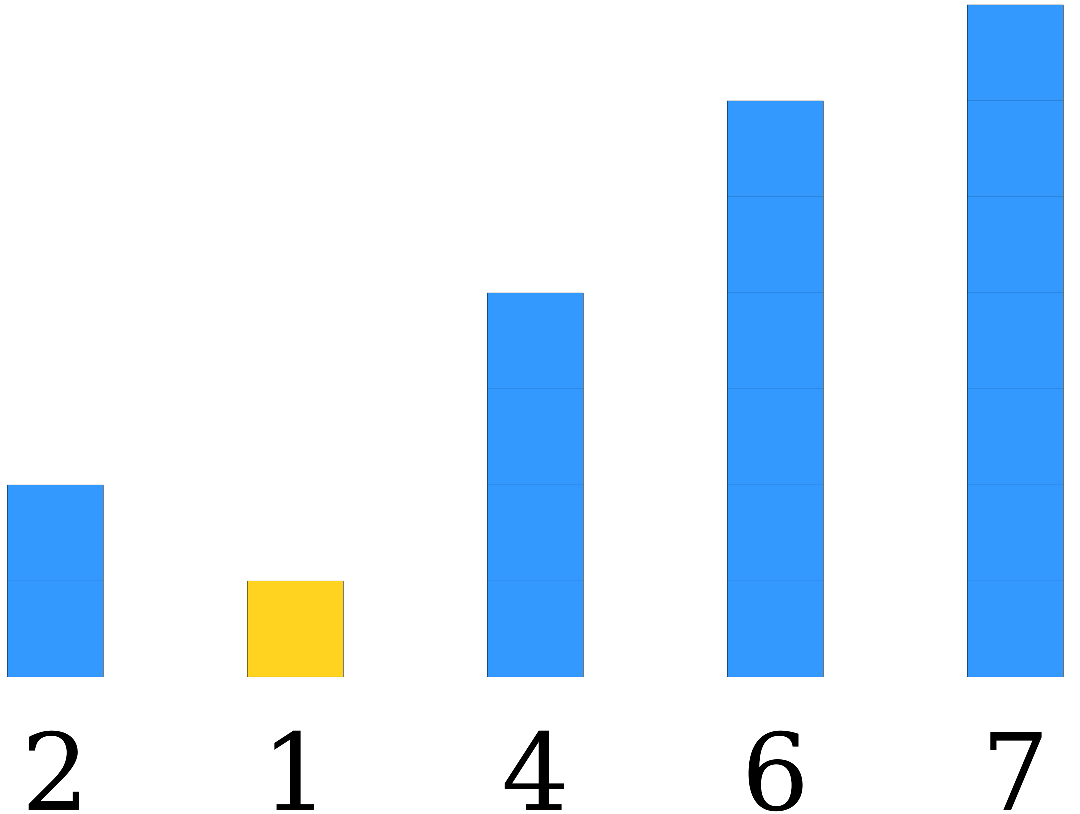
# How Fast is Insertion Sort?



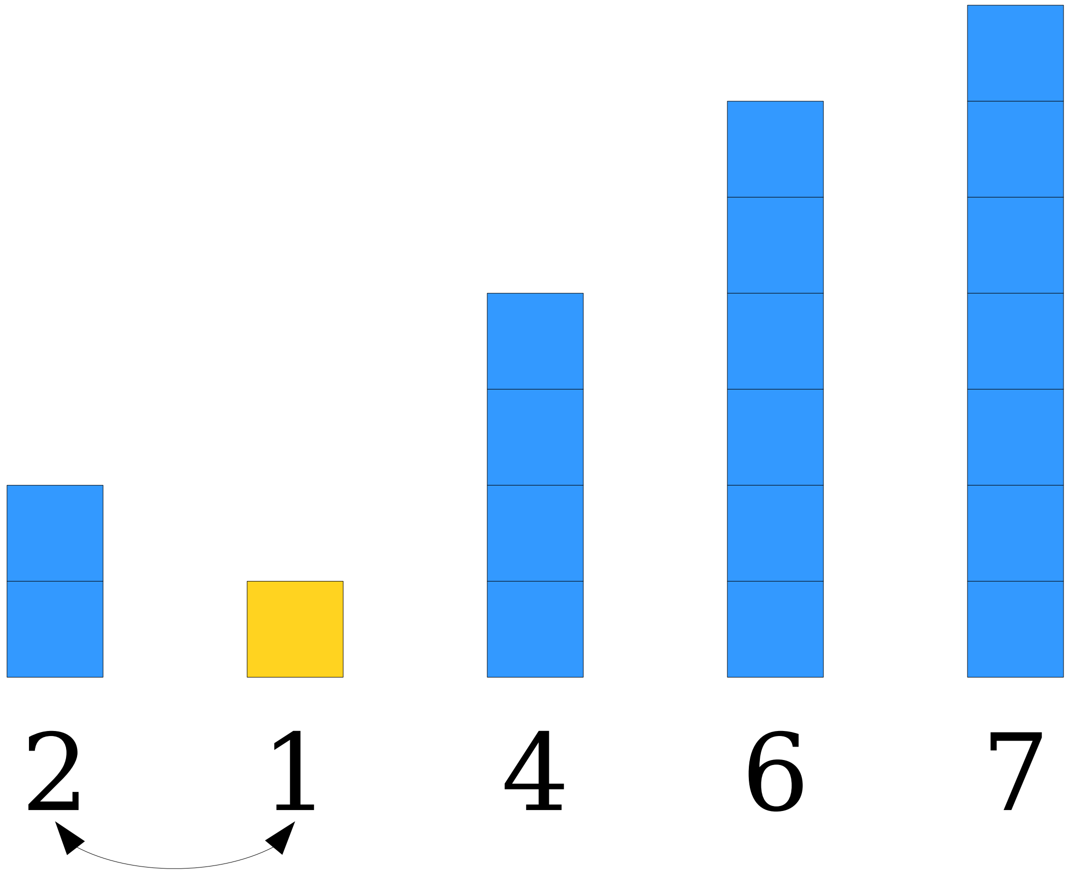
# How Fast is Insertion Sort?



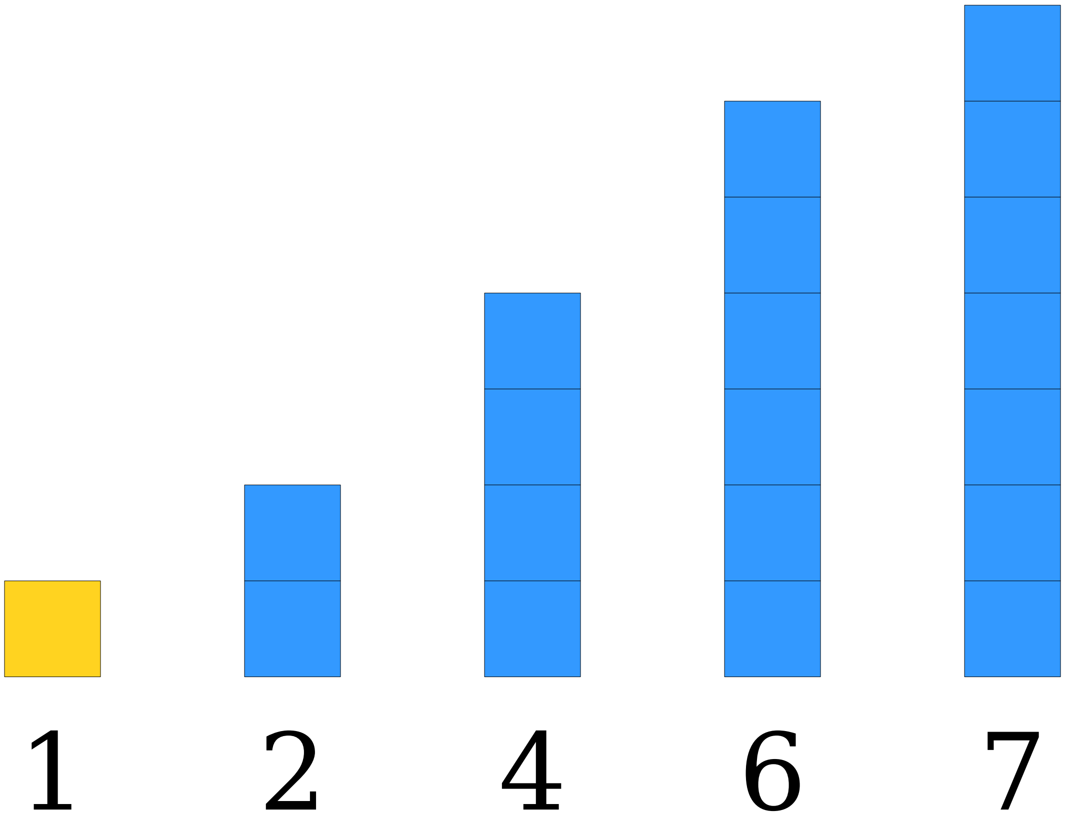
# How Fast is Insertion Sort?



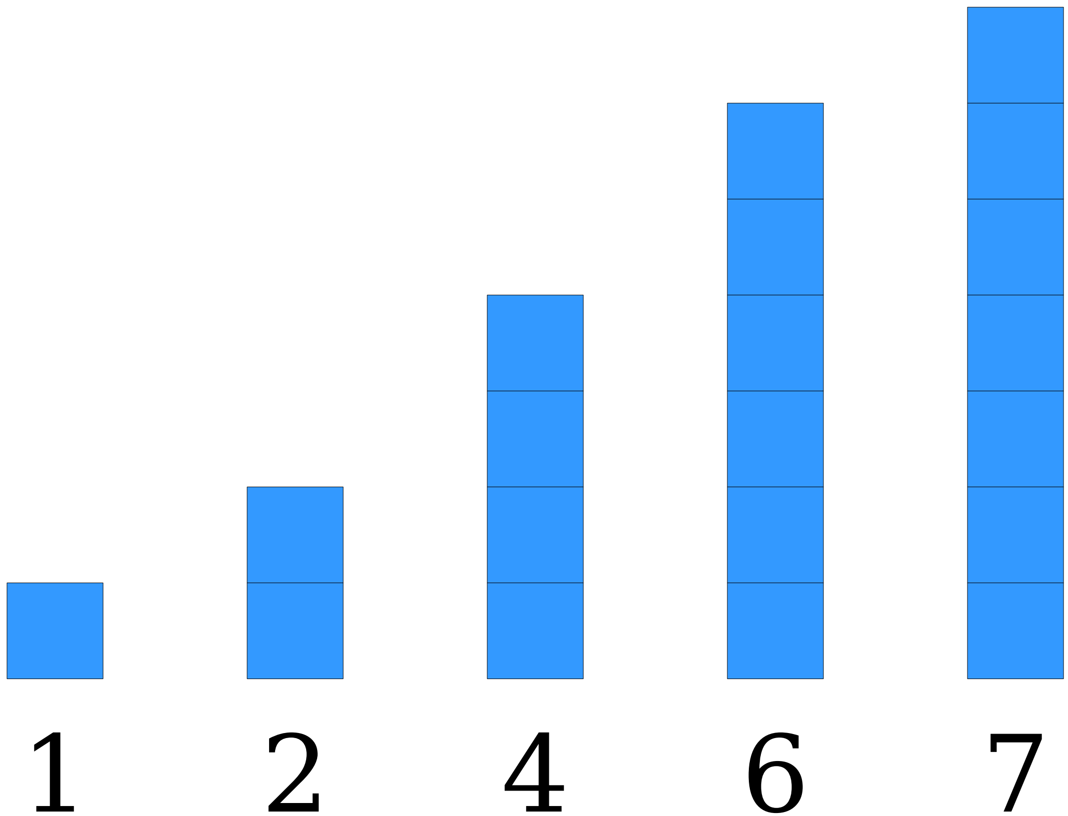
# How Fast is Insertion Sort?



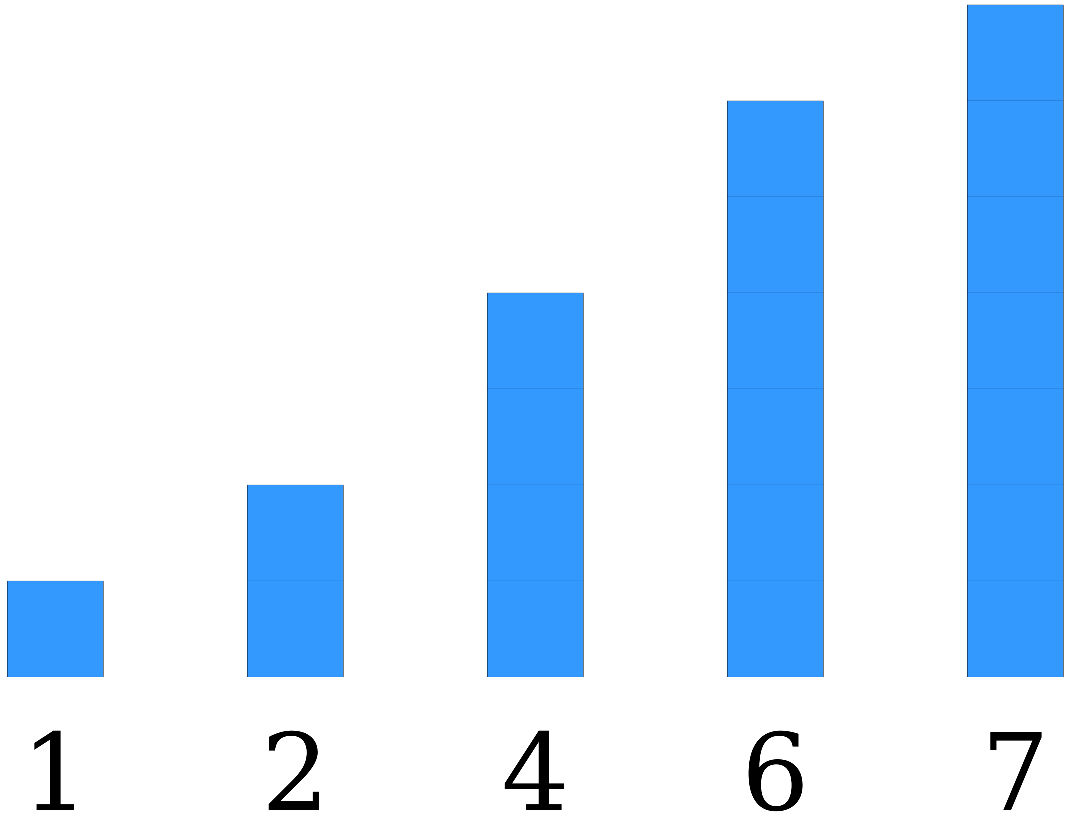
# How Fast is Insertion Sort?



# How Fast is Insertion Sort?



# How Fast is Insertion Sort?




Work Done: $\mathbf{1 + 2 + 3 + \ldots + n}$

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

# Three Analyses

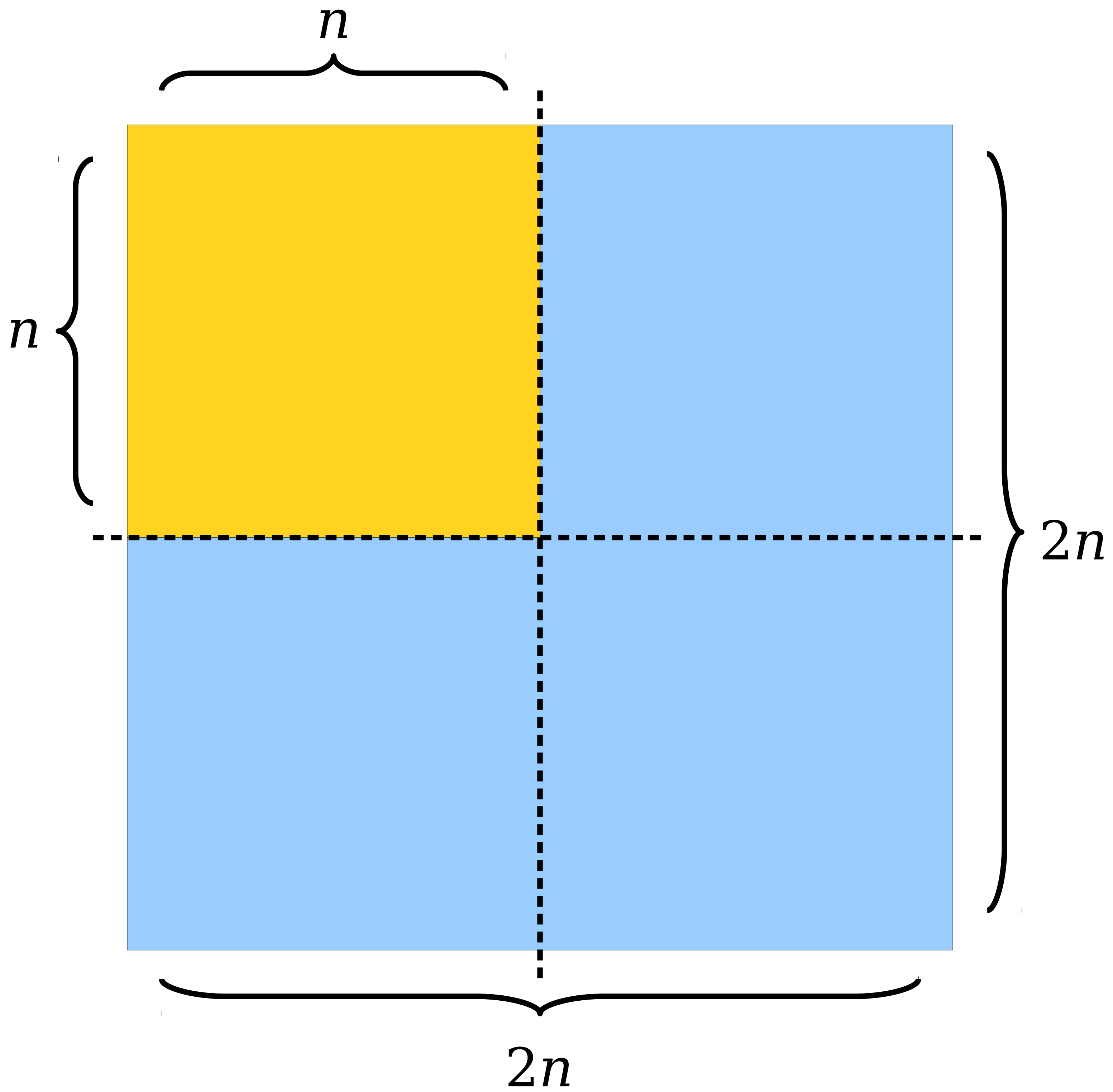
- Worst-Case Analysis
  - What's the *worst* possible runtime for the algorithm?
  - Useful for “sleeping well at night.”
- Best-Case Analysis
  - What's the *best* possible runtime for the algorithm?
  - Useful to see if the algorithm performs well in some cases.
- Average-Case Analysis
  - What's the *average* runtime for the algorithm?
  - Far beyond the scope of this class; take CS109, CS161, or CS265 for more information!

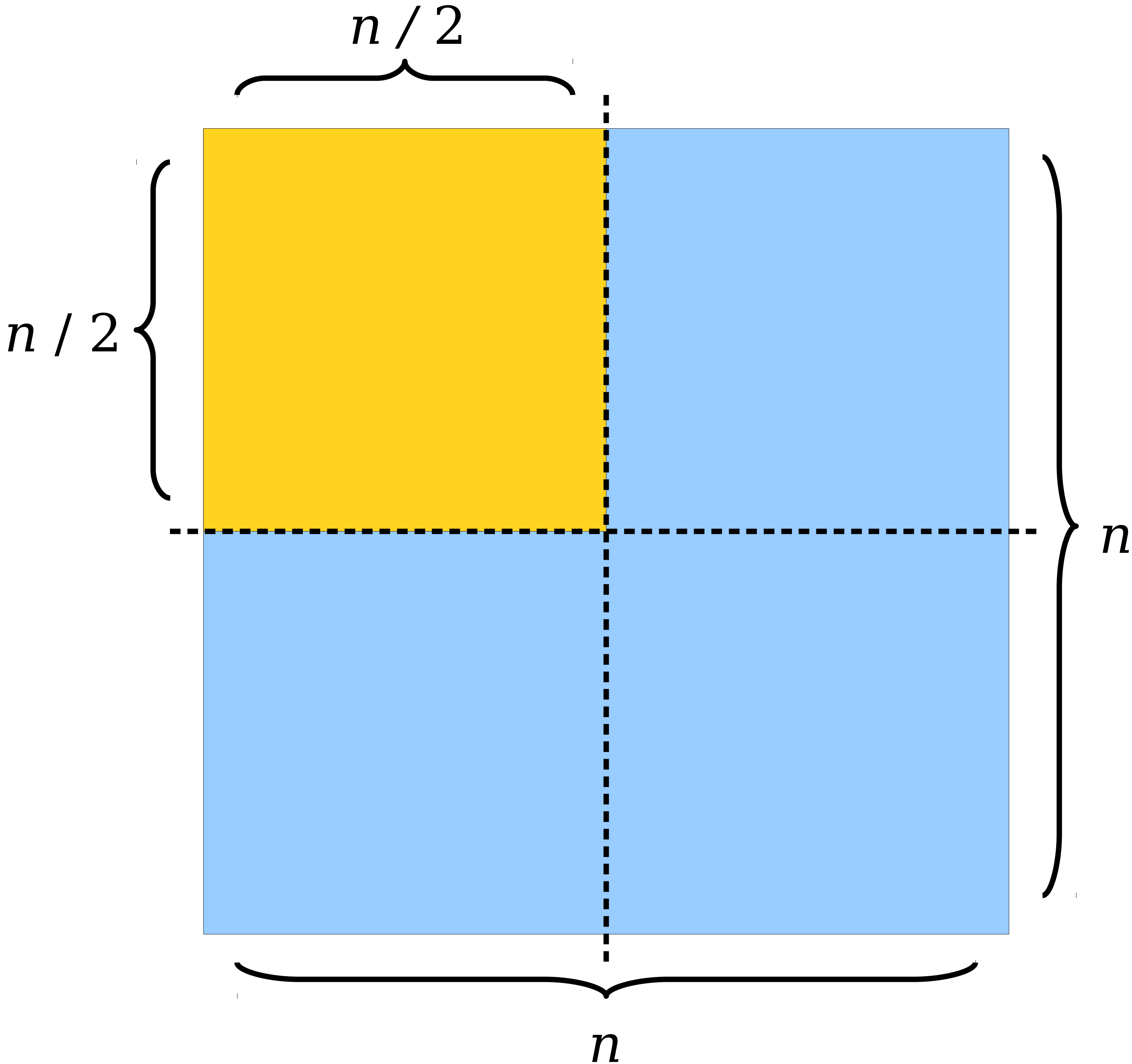
# The Complexity of Insertion Sort

- In the best case (the array is sorted), insertion takes time **$O(n)$**.
- In the worst case (the array is reverse-sorted), insertion sort takes time **$O(n^2)$**.
- ***Fun fact:*** Insertion sorting an array of random values takes, on average, **$O(n^2)$** time.
  - Curious why? Come talk to me after class!

How do selection sort and insertion sort compare against one another?

# Building a Better Sorting Algorithm


n
n
2n
2n


n / 2
n / 2
n
n

# 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($n^2$)

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

¼T($n$)

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

¼T($n$)

# 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 |

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

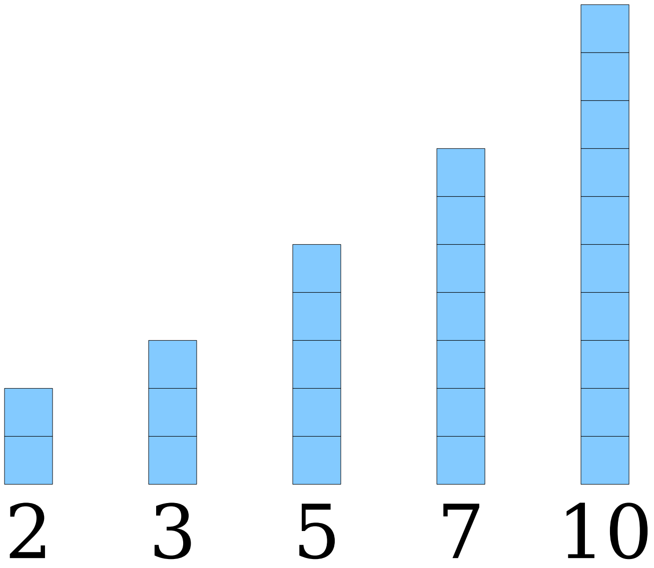
$$2 \cdot \tfrac{1}{4}T(n) = \tfrac{1}{2}T(n)$$

With an O($n^2$)-time sorting algorithm, it takes twice as long to sort the whole array as it does to split the array in half and sort each half.

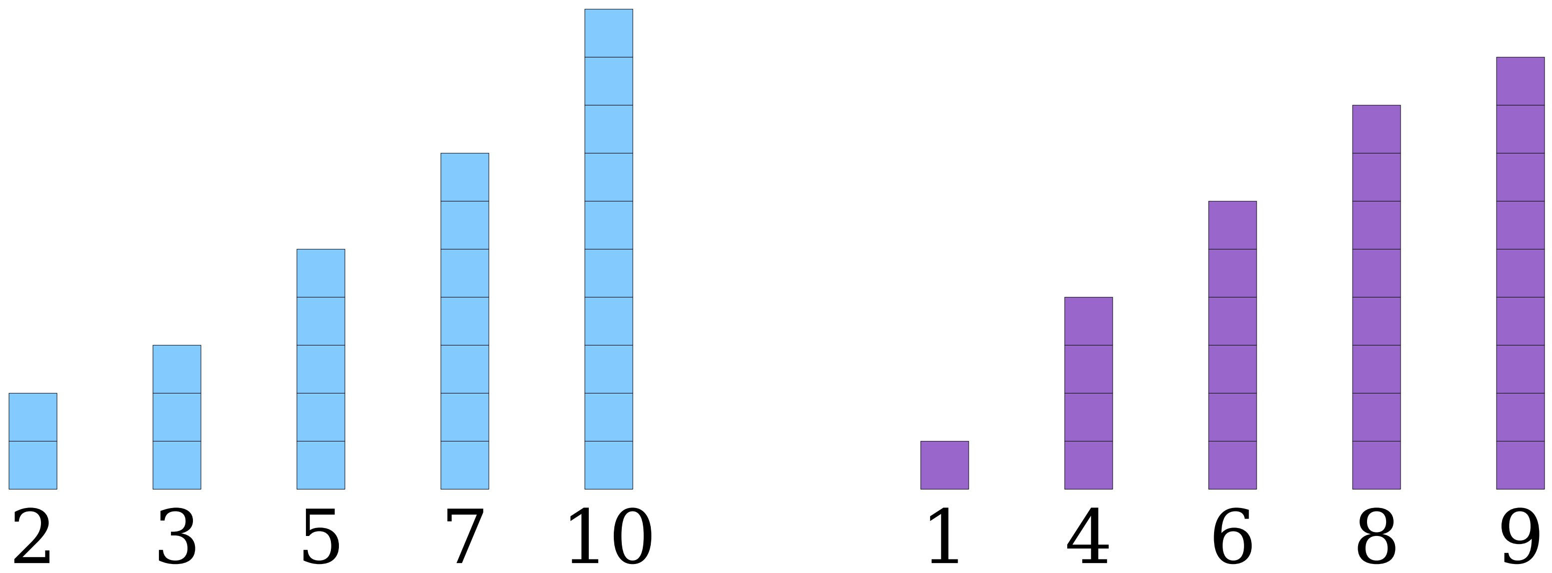
Can we exploit this?

# The Key Insight: ***Merge***

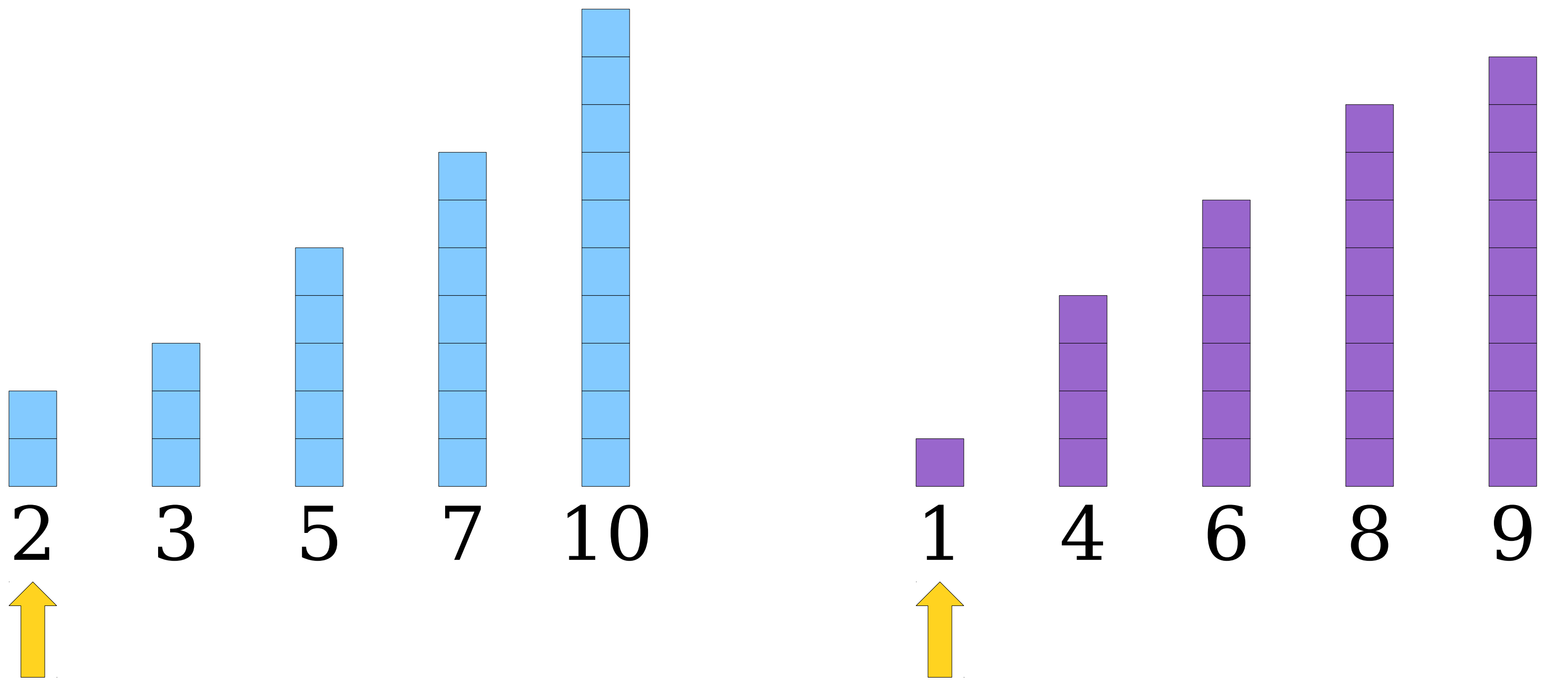
# The Key Insight: ***Merge***



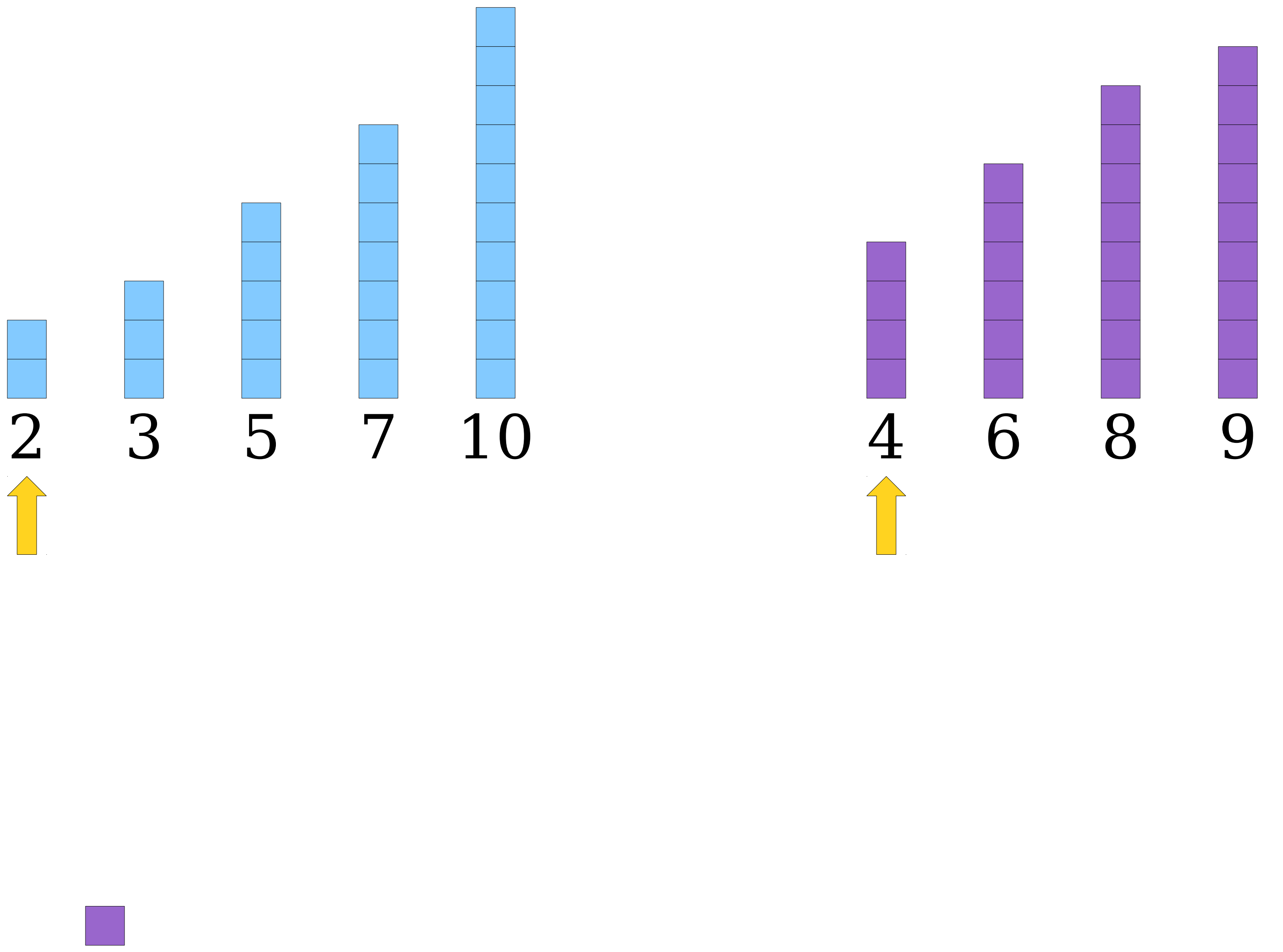
# The Key Insight: ***Merge***



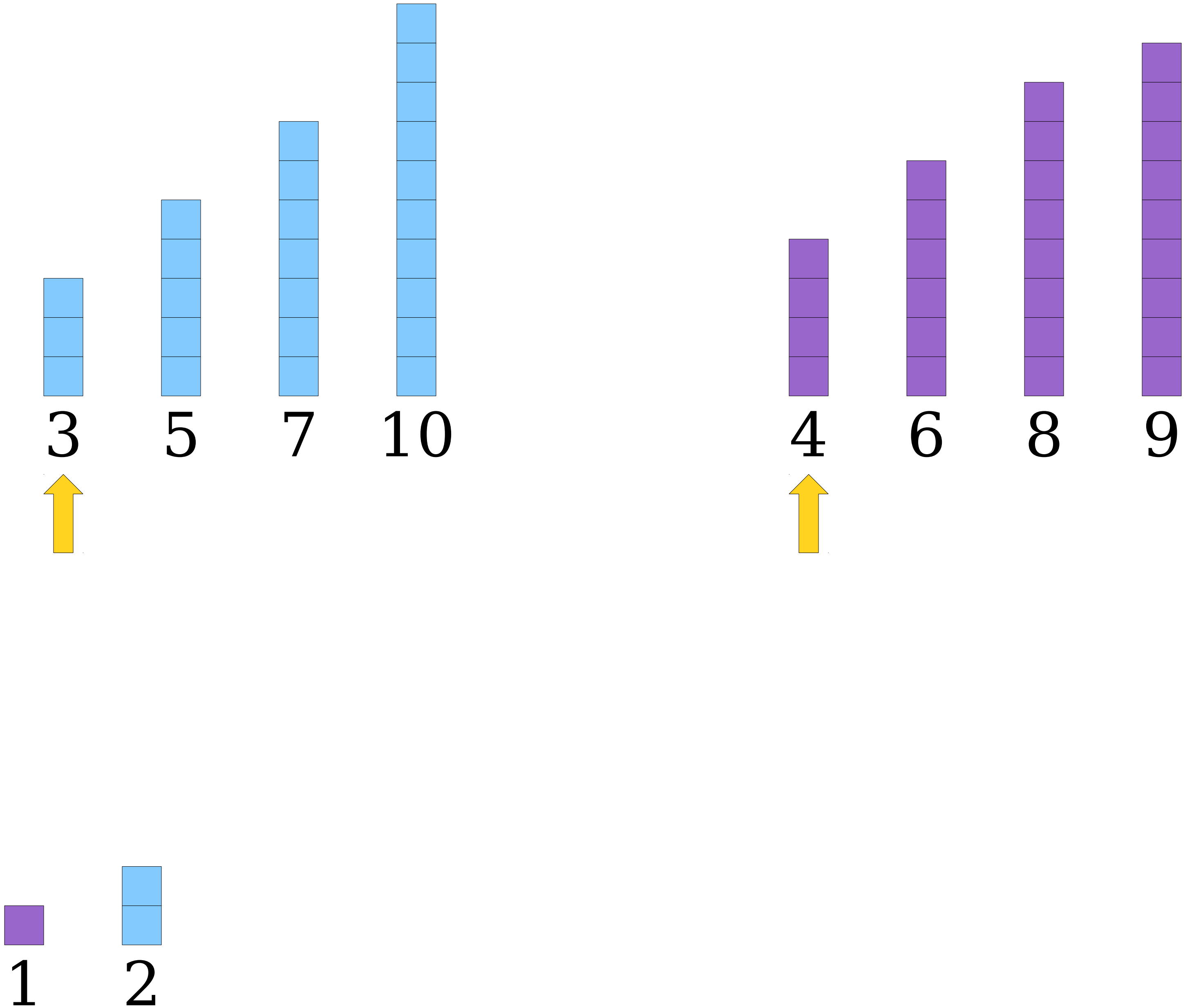
# The Key Insight: ***Merge***



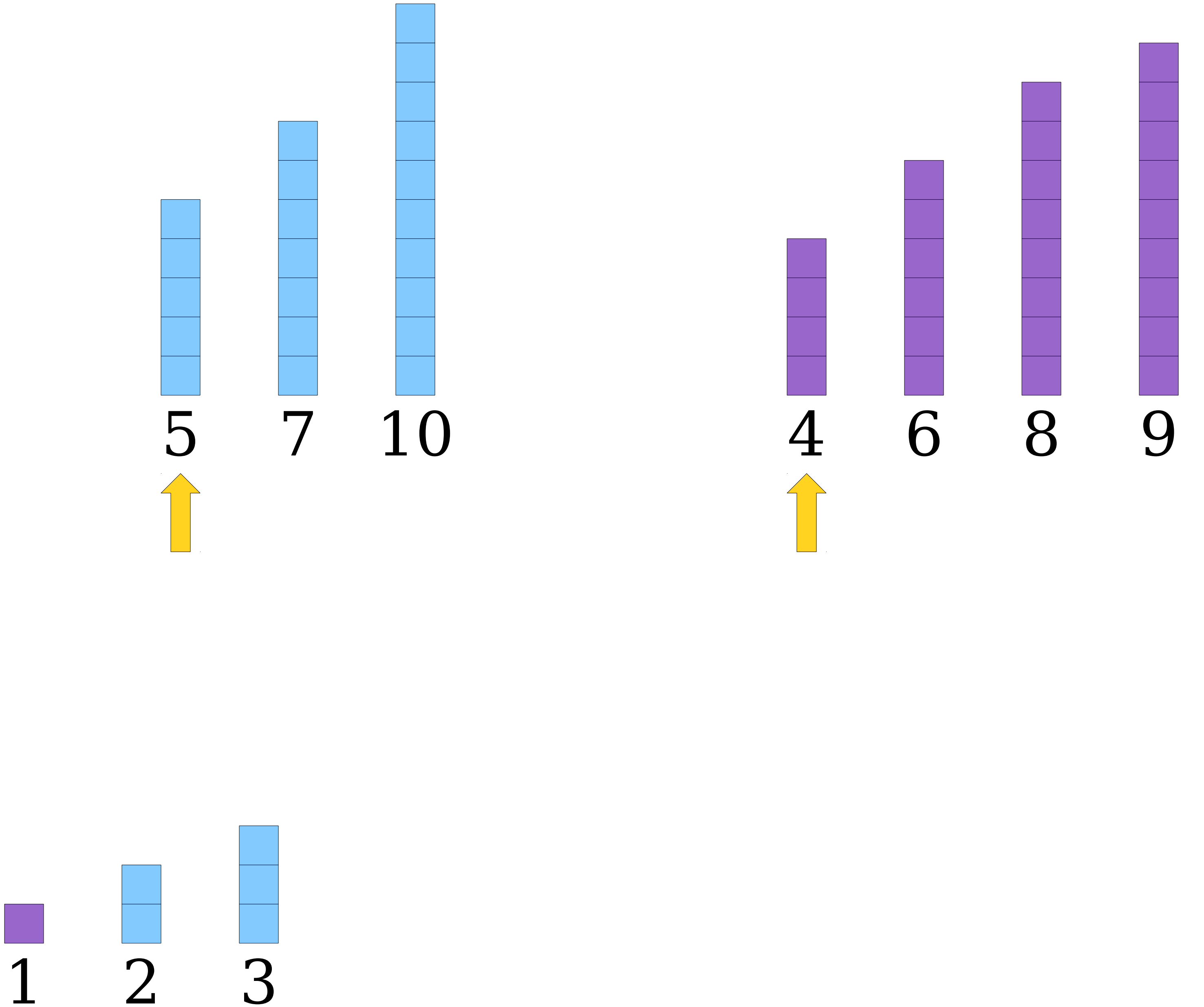
# The Key Insight: ***Merge***



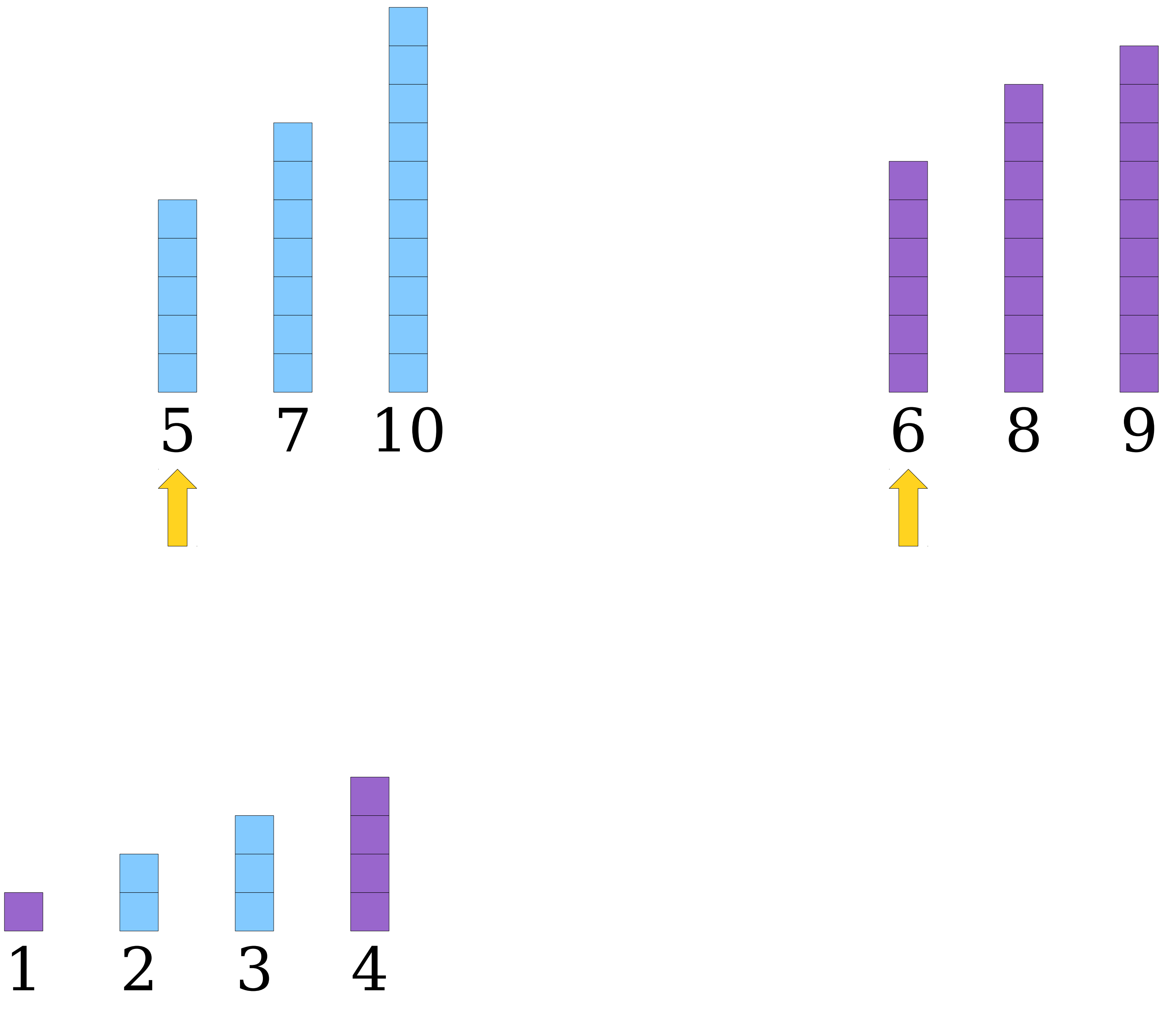
# The Key Insight: ***Merge***



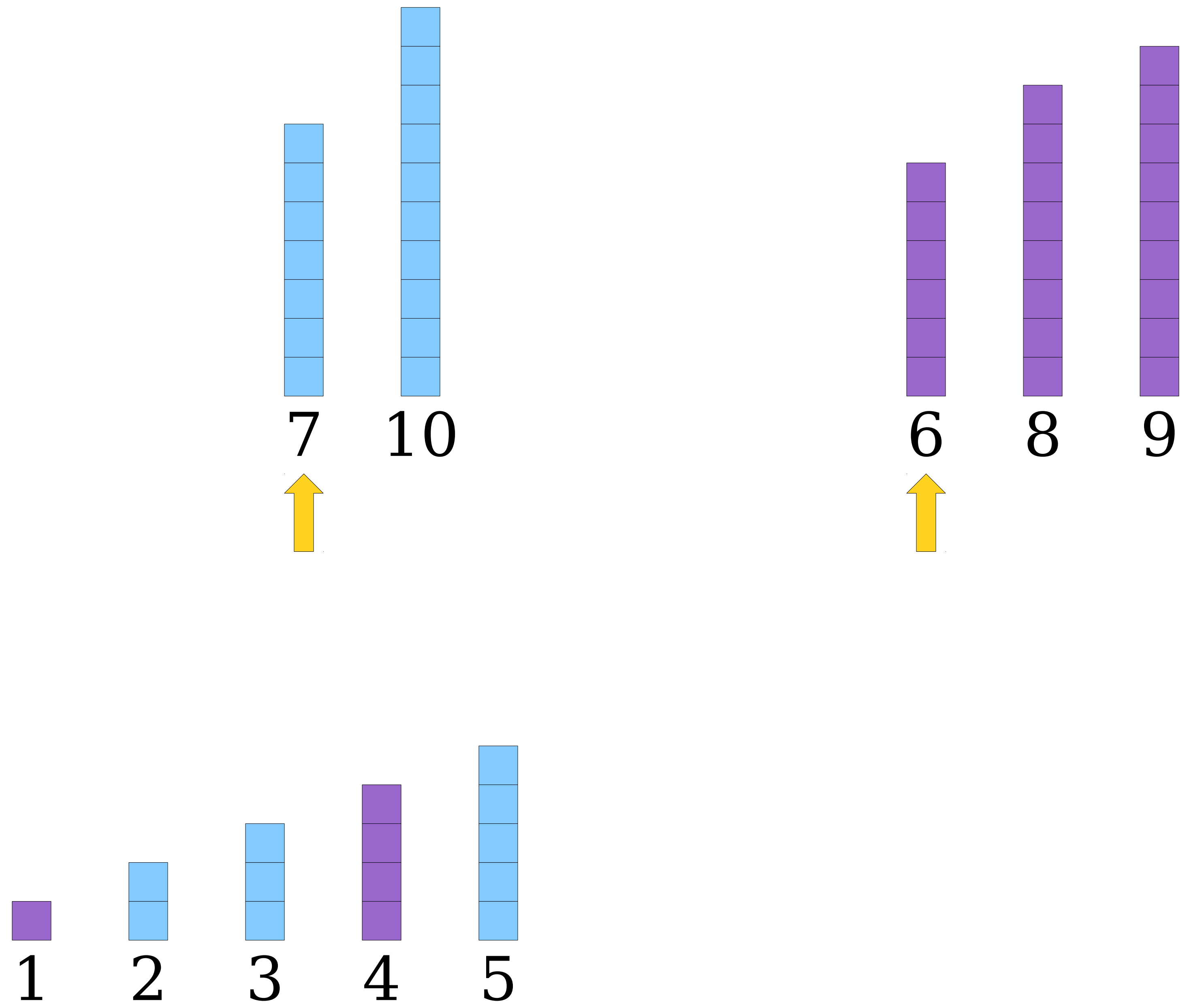
# The Key Insight: ***Merge***



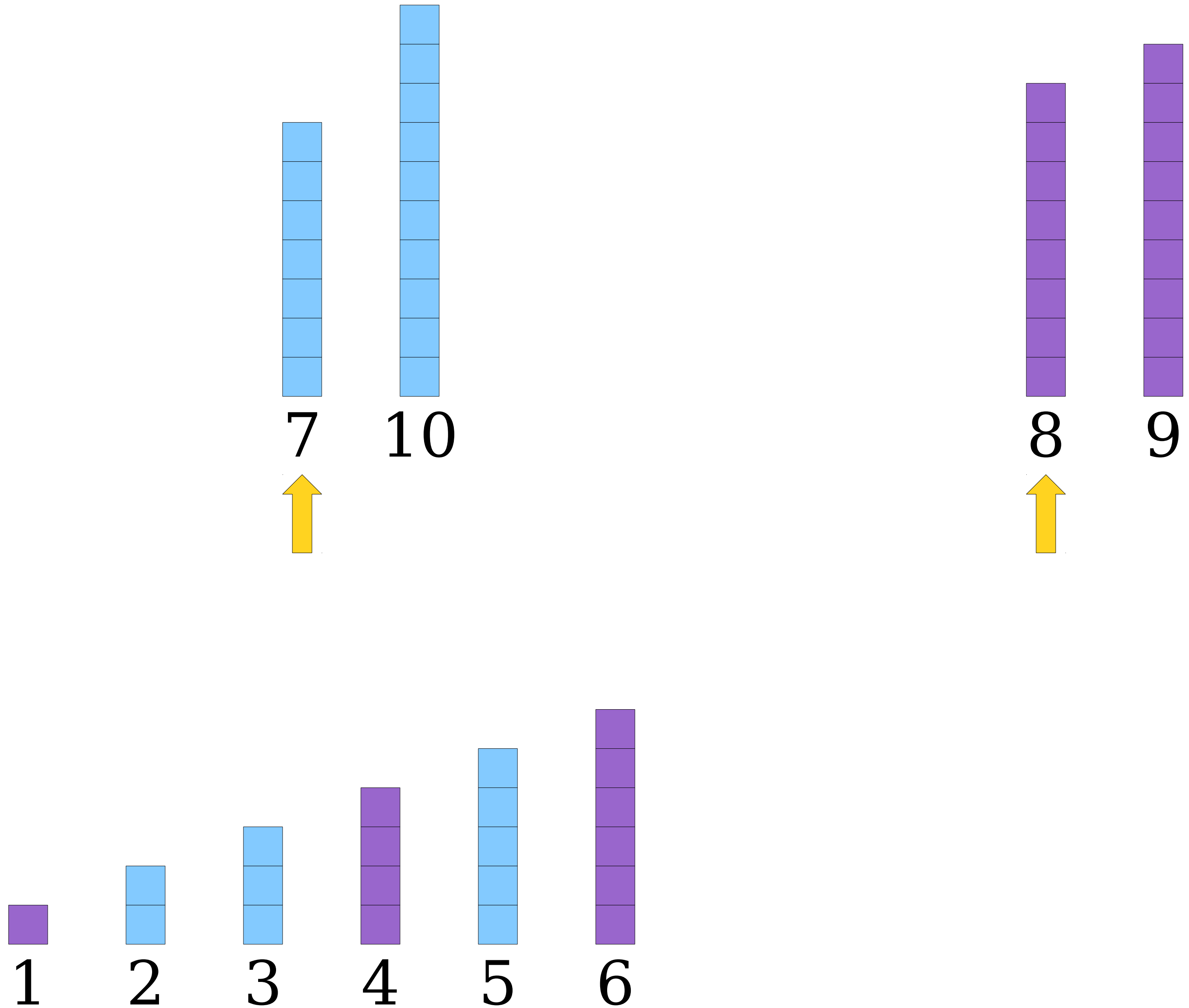
# The Key Insight: ***Merge***



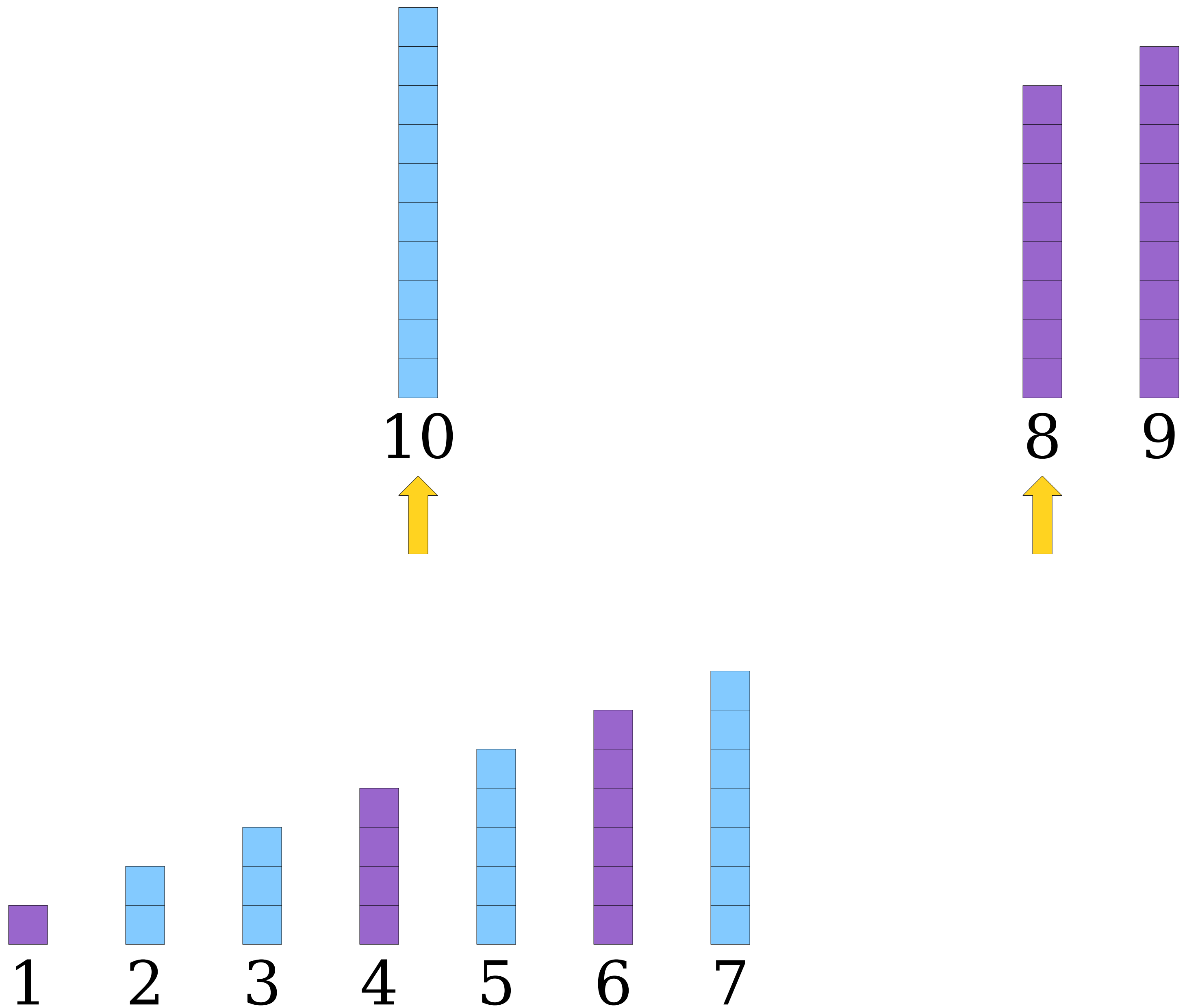
# The Key Insight: ***Merge***



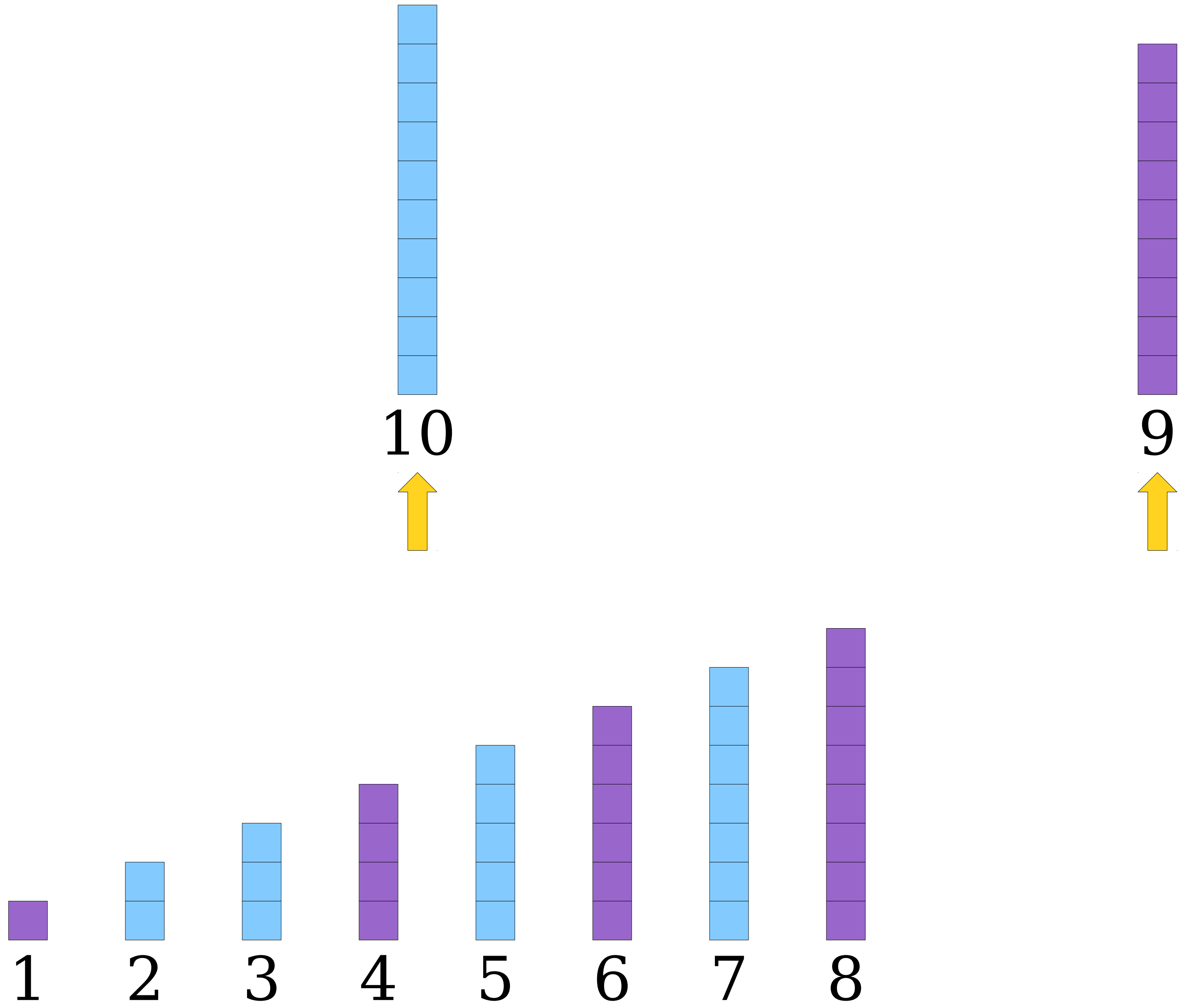
# The Key Insight: ***Merge***



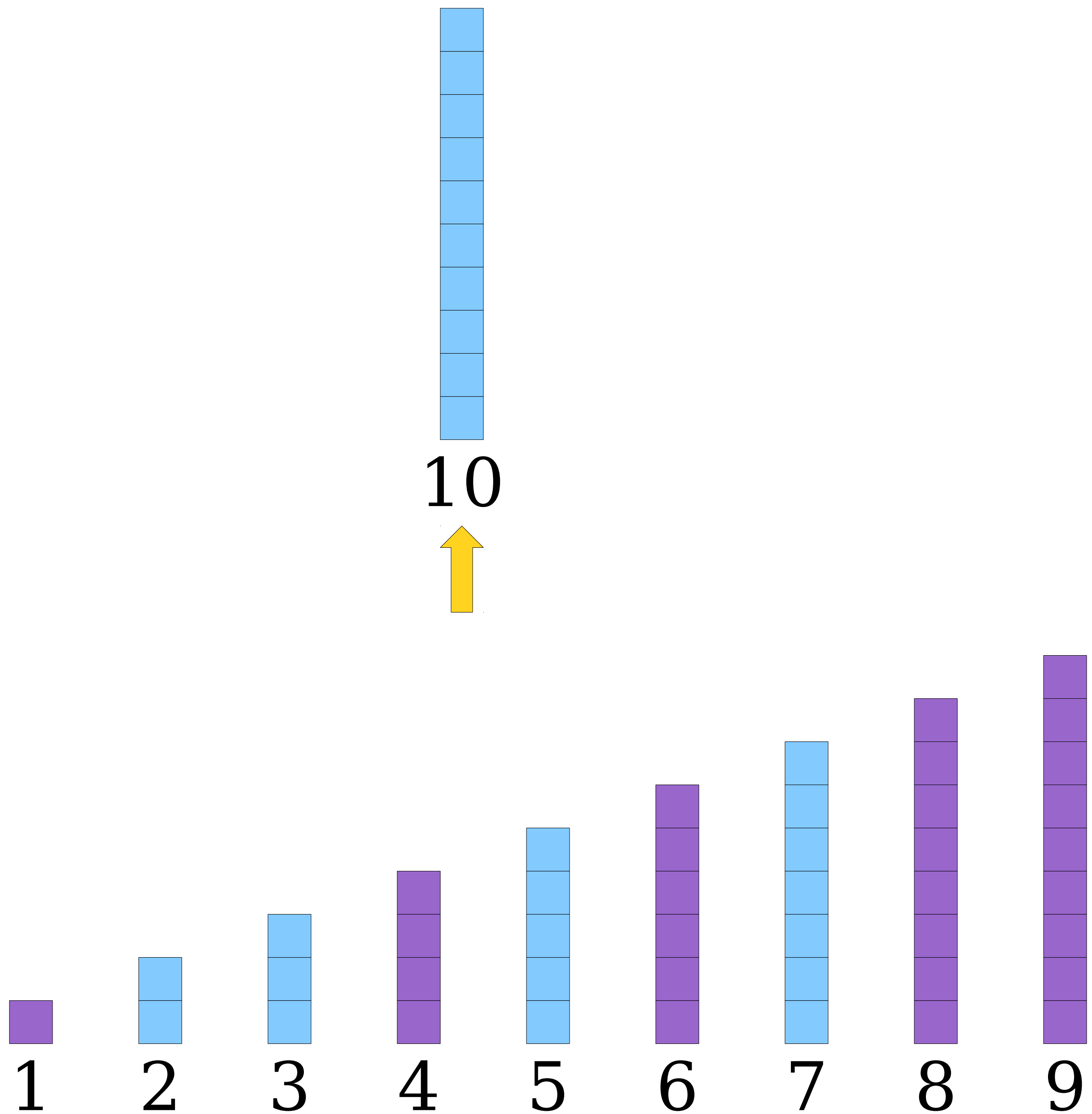
# The Key Insight: ***Merge***



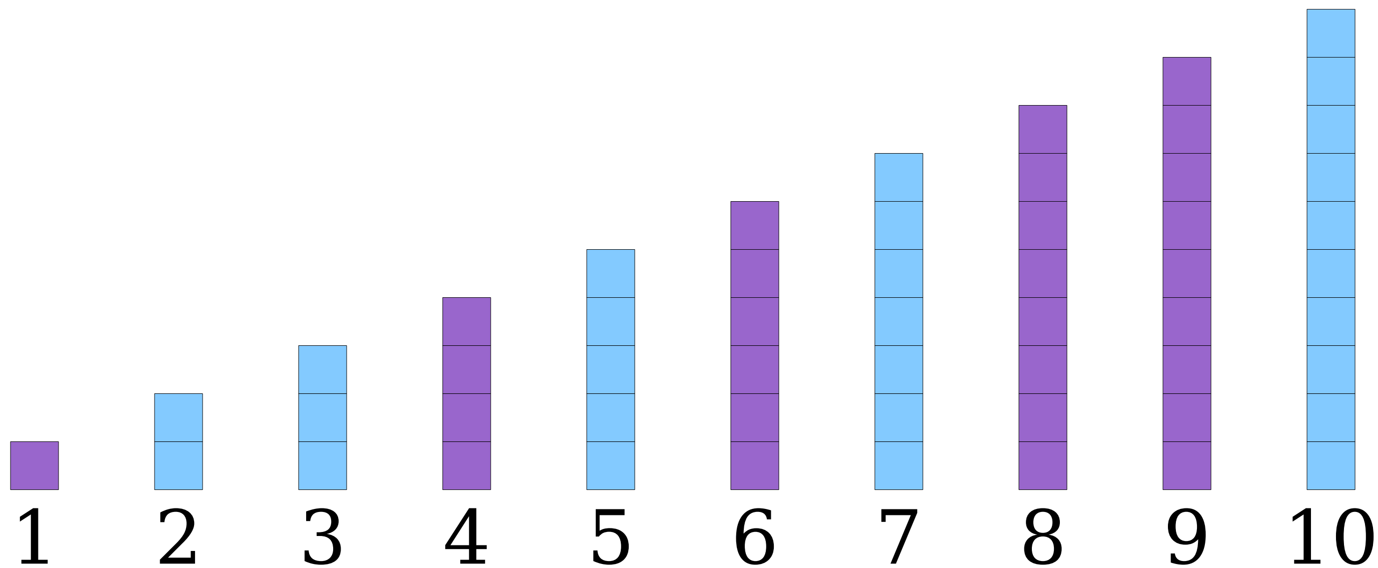
# The Key Insight: ***Merge***



# The Key Insight: ***Merge***



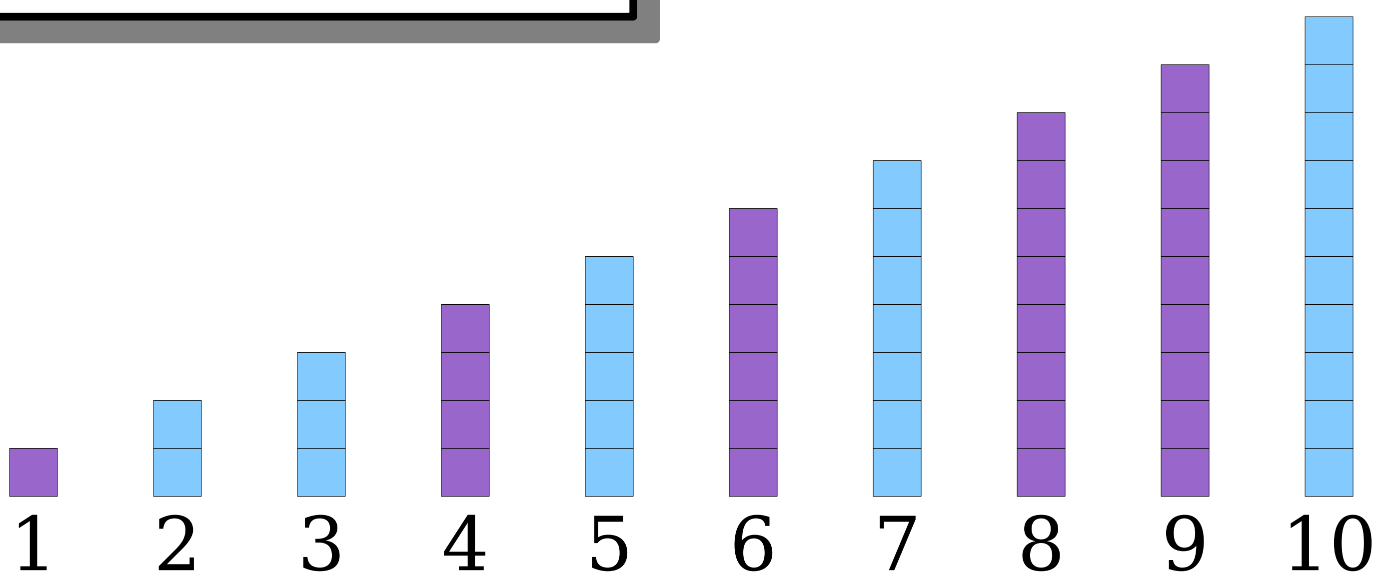
# The Key Insight: ***Merge***



# The Key Insight: ***Merge***

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

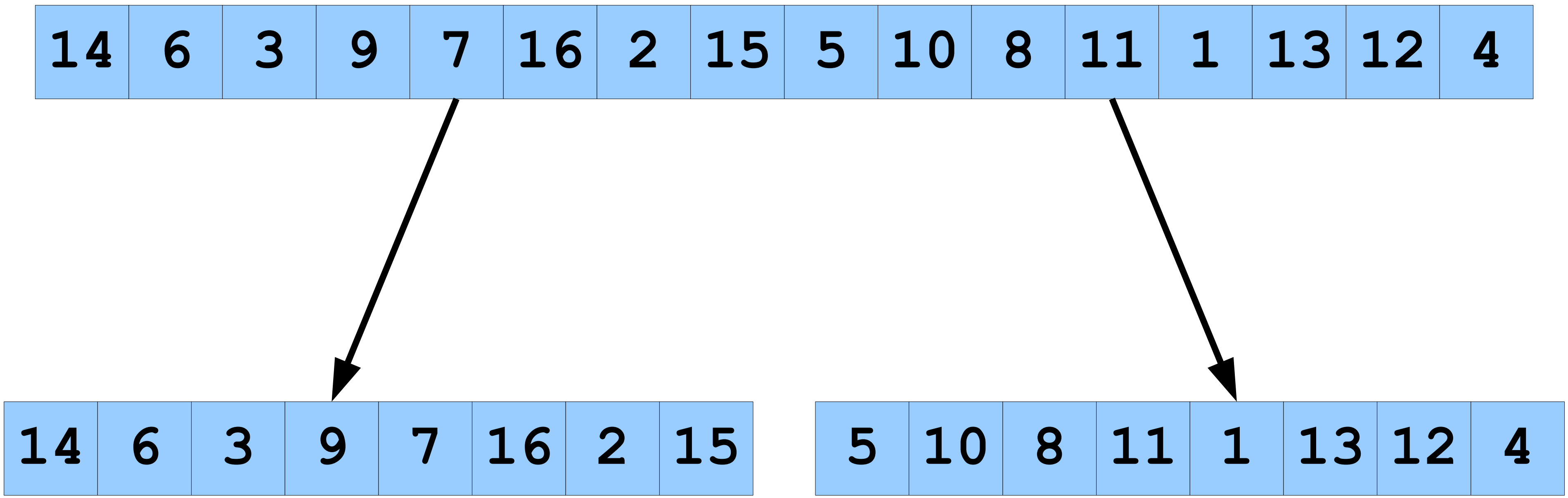
If there are *n* total elements, this algorithm runs in time **O(*n*)**.



# The Key Insight: ***Merge***

- The ***merge*** algorithm takes in two sorted lists and combines them into a single sorted list.
- While both lists are nonempty, compare their first elements. Remove the smaller element and append it to the output.
- Once one list is empty, add all elements from the other list to the output.
- ***Note:*** Removing the first element of a `Vector` takes time O($n$), so you can't implement this algorithm by repeatedly removing the first element of one of the `Vector`s. See if you can find another approach for doing this.

# “Split Sort”




1. Split the input in half.

# “Split Sort”

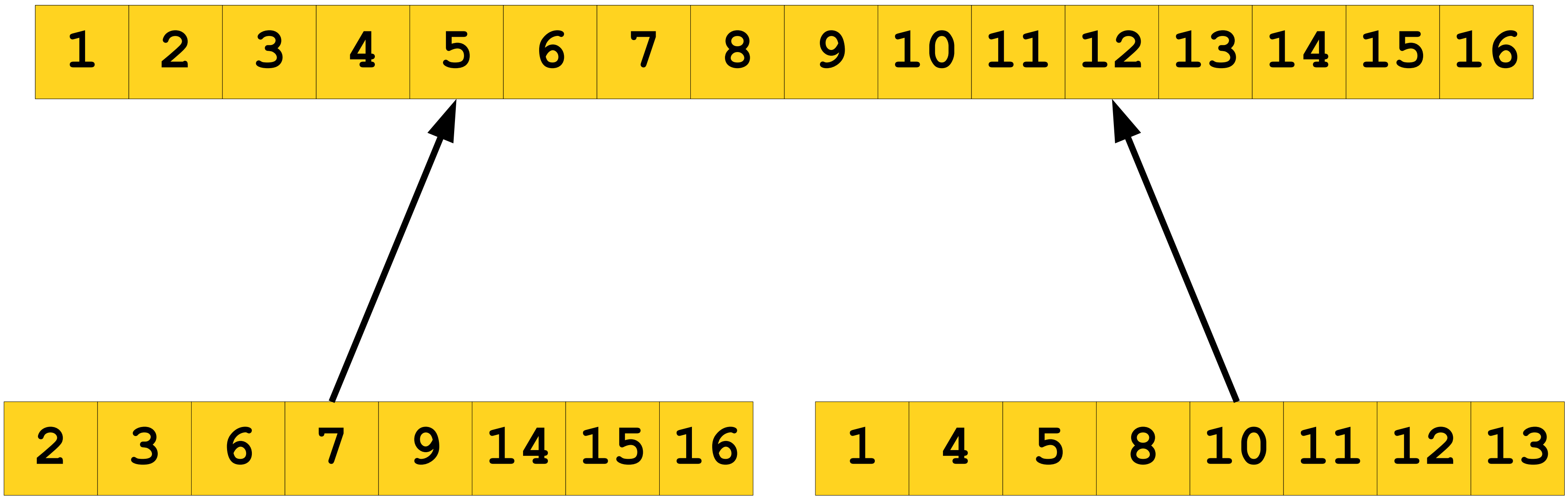
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 14 | 6 | 3 | 9 | 7 | 16 | 2 | 15 | 5 | 10 | 8 | 11 | 1 | 13 | 12 | 4 |

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

    /* Sort each half. */
    insertionSort(left);
    insertionSort(right);

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

Takes O($n$) time, since we copy all $n$ elements into new `Vectors`.

Takes O($n^2$) time, but about half as much as what we did before.

Takes O($n$) time.

***Prediction:*** This should still take time O($n^2$), but be about twice as fast as insertion sort.

# Next Time

- ***Mergesort***
  - A beautiful, elegant sorting algorithm.
- ***Analyzing Mergesort***
  - An unusual runtime analysis.
- ***Hybrid Sorting Algorithms***
  - Improving on mergesort.
- ***Binary Search***
  - Finding things fast!