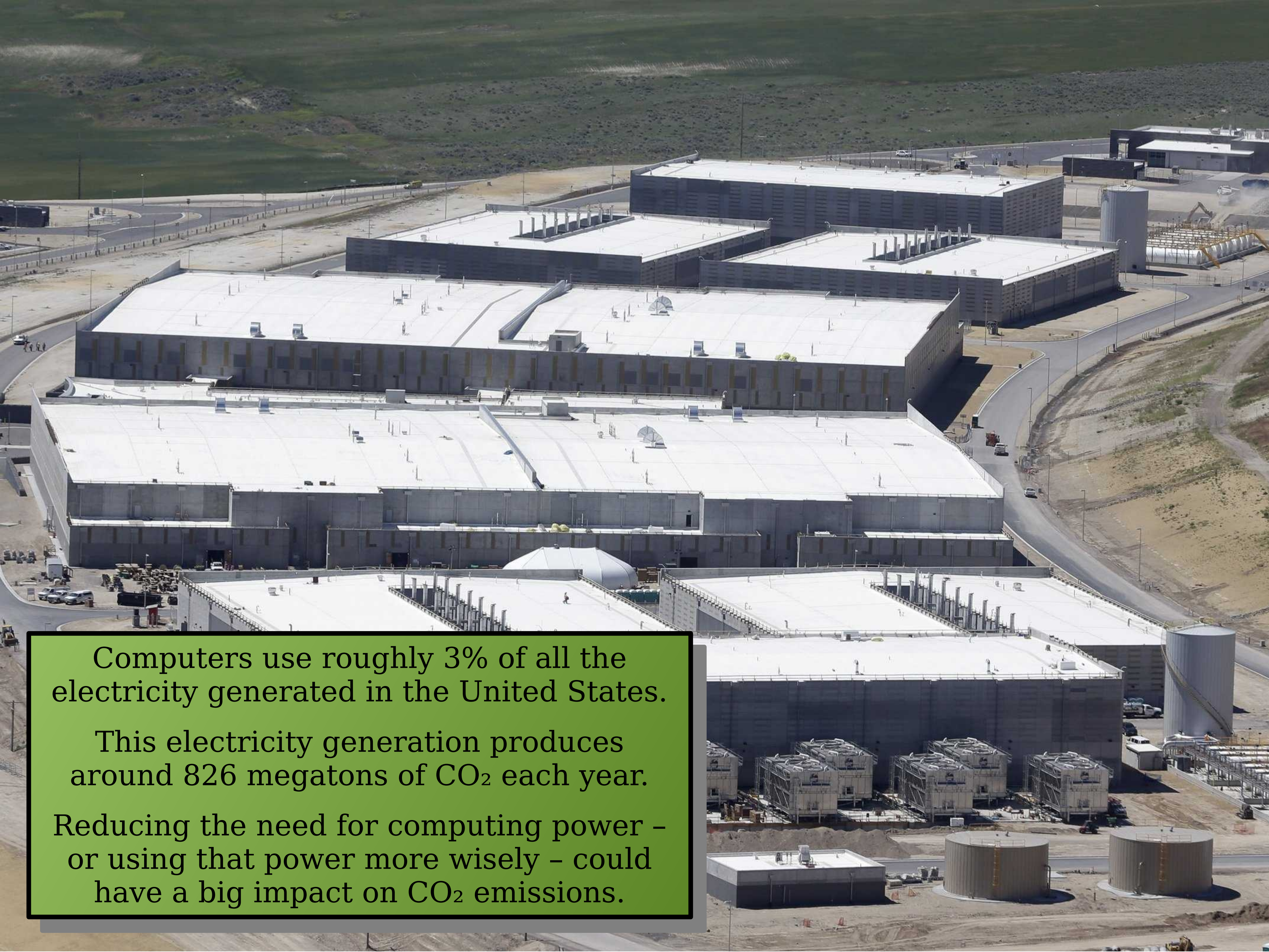
Algorithmic Analysis and Sorting

Part One



Computers use roughly 3% of all the electricity generated in the United States.

This electricity generation produces around 826 megatons of CO_2 each year.

Reducing the need for computing power – or using that power more wisely – could have a big impact on CO_2 emissions.

Fundamental Question:

How do we measure efficiency?

One Idea: ***Runtime***

Runtime is Noisy

- Runtime is highly sensitive to ***which computer you're using***.
- Runtime is highly sensitive to ***which inputs you're testing***.
- Runtime is highly sensitive to ***external factors***.

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

Work Done: At most $k_0 n + k_1$

Big Observations

- If our goal is to extrapolate out the runtime, we don’t need to know the constants in advance. We can figure them out by running the code.
- For “sufficiently large” inputs, only the dominant term matters.
  - For both $4n + 1000$ and $n + 137$, for very large n most of the runtime is explained by n.
- Is there a concise way of describing this?

Big-O

Big-O Notation

- Ignore *everything* except the dominant growth term, including constant factors.
- Examples:
  - $4n + 4 = \mathbf{O(n)}$
  - $137n + 271 = \mathbf{O(n)}$
  - $n^2 + 3n + 4 = \mathbf{O(n^2)}$
  - $2^n + n^3 = \mathbf{O(2^n)}$

For the mathematically inclined:

$$f(n) = O(g(n)) \text{ if}$$
$$\exists n_0 \in \mathbb{R}.\ \exists c \in \mathbb{R}.\ \forall n \geq n_0.\ f(n) \leq c|g(n)|$$

Algorithmic Analysis with Big-O

Algorithmic Analysis with Big-O

```
double average(const Vector<int>& vec) {
  double total = 0.0;
  for (int i = 0; i < vec.size(); i++) {
      total += vec[i];
  }

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

Algorithmic Analysis with Big-O

```
double average(const Vector<int>& vec) {
  double total = 0.0;
  for (int i = 0; i < vec.size(); i++) {
      total += vec[i];
  }

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

Algorithmic Analysis with Big-O

```
double average(const Vector<int>& vec) {
  double total = 0.0;
  for (int i = 0; i < vec.size(); i++) {
      total += vec[i];
  }

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

O(*n*)

O(n) means “the runtime is proportional to the size of the input.” We’d say that this code runs in ***linear time***.

A More Interesting Example

A More Interesting Example

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

How do we analyze this?

Types of Analysis

- 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!

Types of Analysis

- 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, CS365, or CS369N for more information!

Being Pessimistic

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

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

Types of Analysis

- 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!

Types of Analysis

- 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!

Three Cheers for Optimism!

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

O(1) means "the runtime doesn't depend on the size of the input." In the best case, this code runs in ***constant time***.

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

What Can Big-O Tell Us?

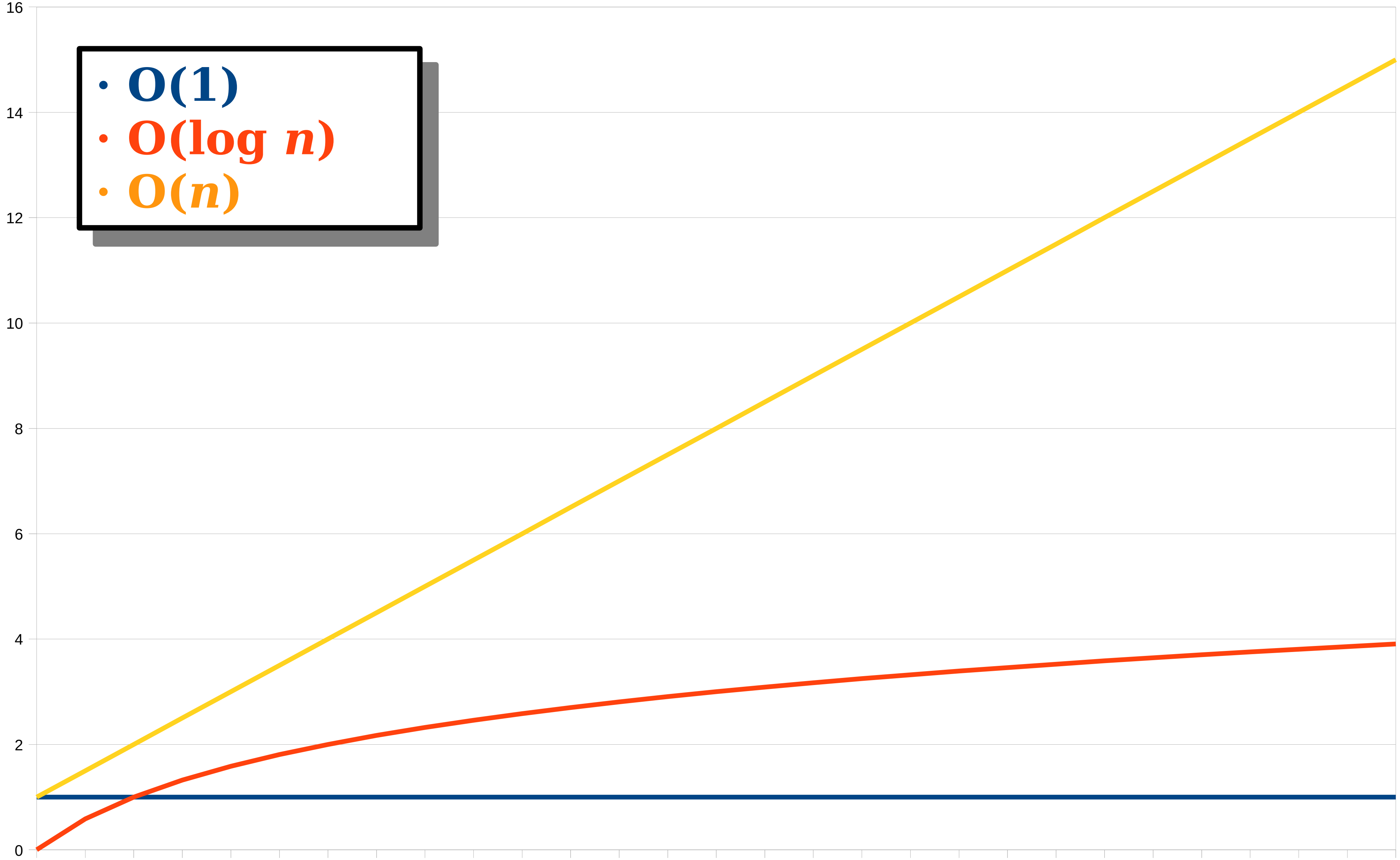
- Long-term behavior of a function.
  - If algorithm A has runtime O(n) and algorithm B has runtime O(n^2), for very large inputs algorithm A will always be faster.
  - If algorithm A has runtime O(n), for large inputs, doubling the size of the input doubles the runtime.

What *Can't* Big-O Tell Us?

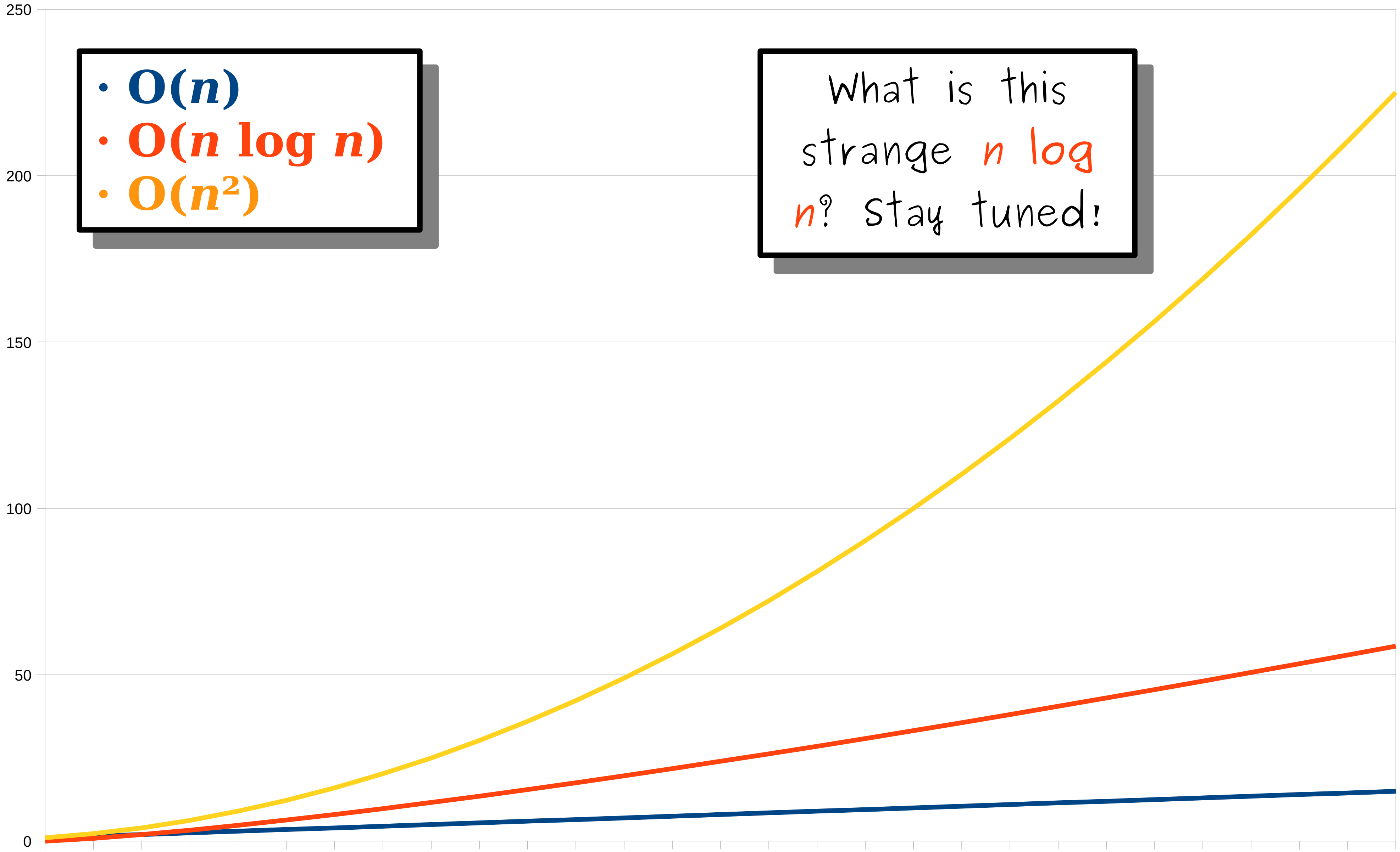
- The actual runtime of a function.
  - $10^{100}n = O(n)$
  - $10^{-100}n = O(n)$
- How a function behaves on small inputs.
  - $n^3 = O(n^3)$
  - $10^6 = O(1)$

Some Standard Runtime Complexities

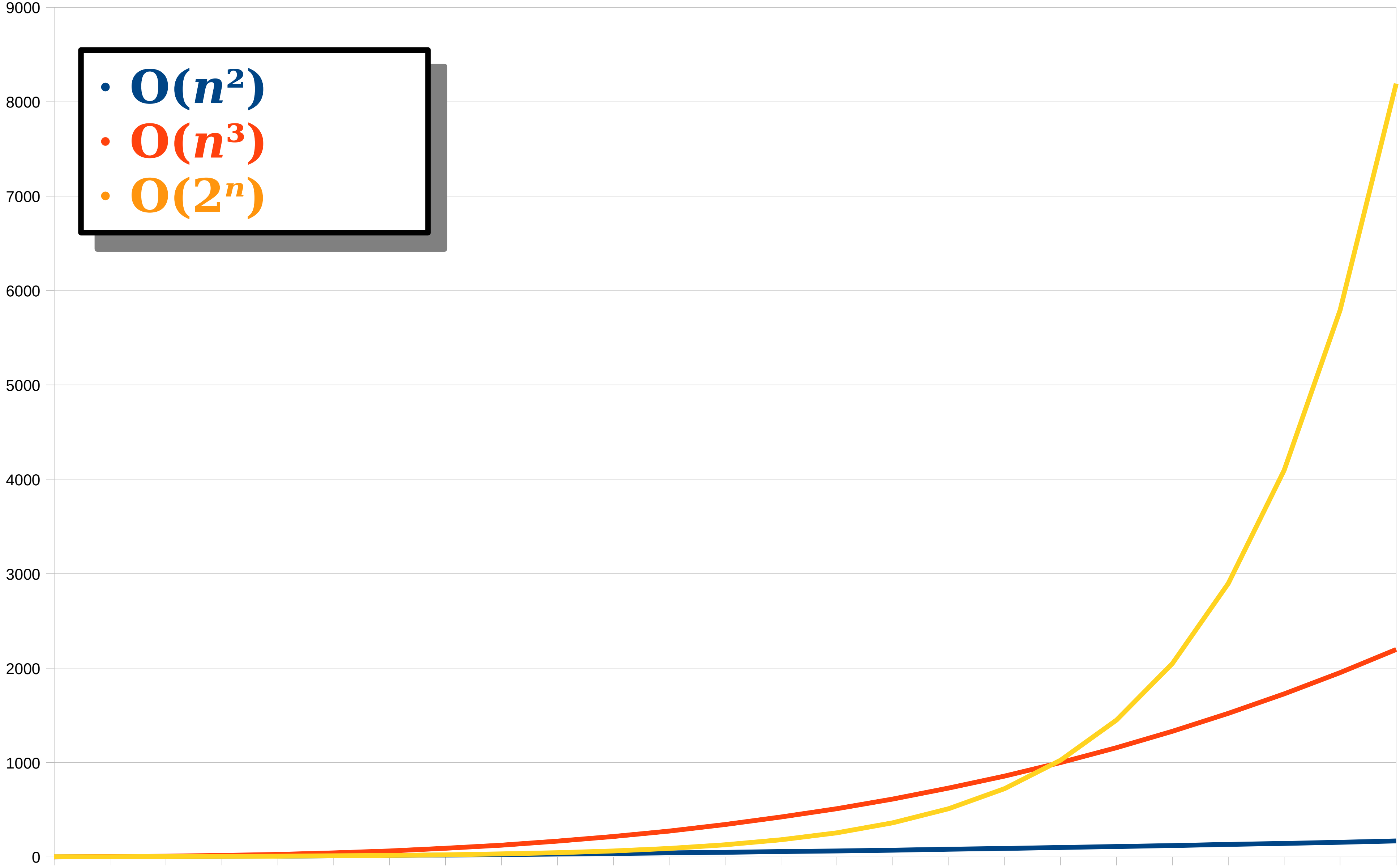
Growth Rates, Part I



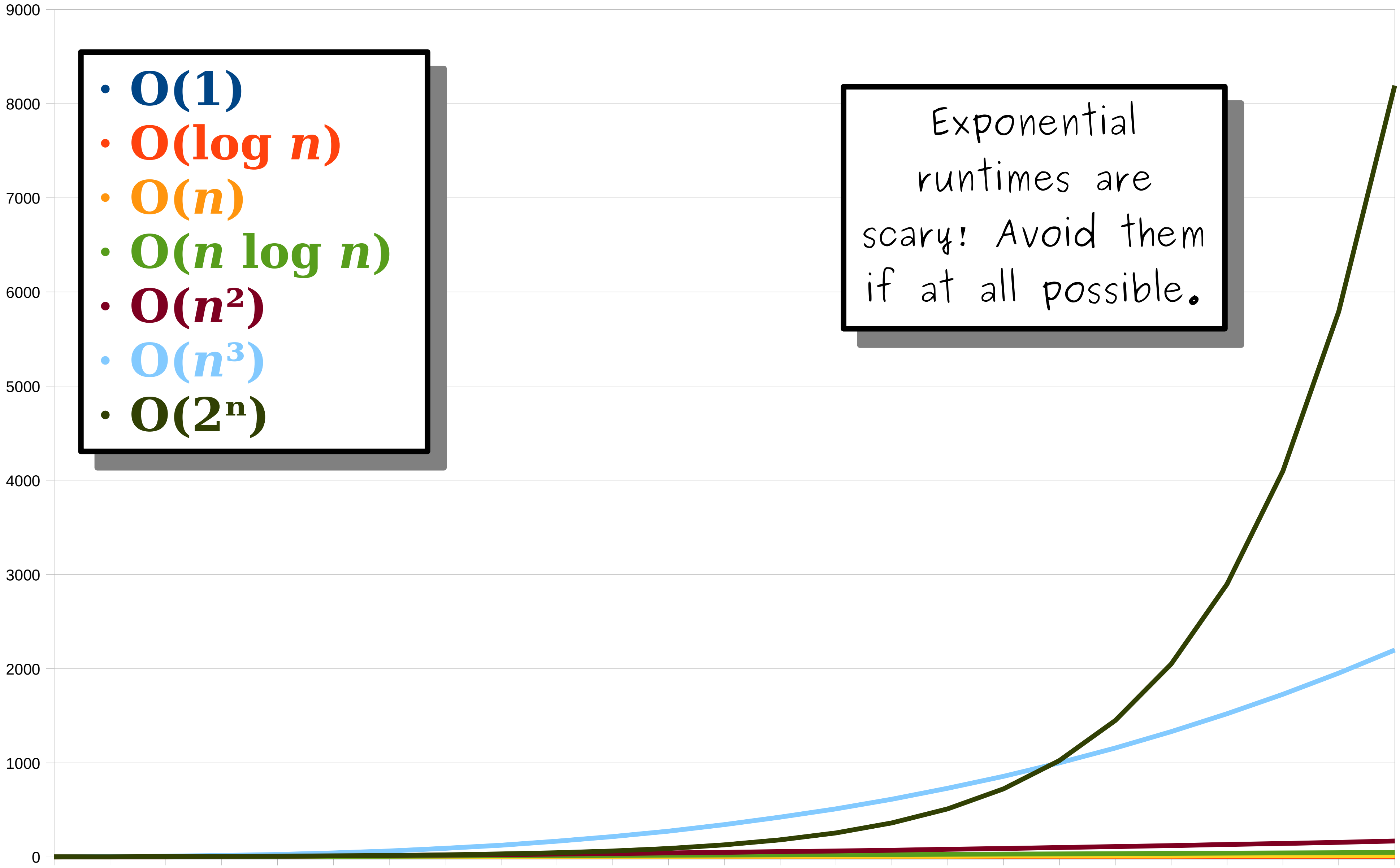
Growth Rates, Part II



Growth Rates, Part III



All Together Now!



Comparison of Runtimes

(assuming 1 operation = 1 nanosecond)

Size	1
1000	1ns
2000	1ns
3000	1ns
4000	1ns
5000	1ns
6000	1ns
7000	1ns
8000	1ns
9000	1ns
10000	1ns
11000	1ns
12000	1ns
13000	1ns
14000	1ns

The Story So Far

- Big-O notation is a quantitative measure of how a function's runtime scales.
- It ignores constants and lower-order terms. Only the fastest-growing terms matter.
- Big-O notation lets us predict how long a function will take to run.
- Big-O notation lets us quantitatively compare algorithms.

Time-Out for Announcements!

Programming Assignments

- Assignment 3 is due on Wednesday.
  - If you're following our timetable, you should be done with the Sierpinski triangle, Human Pyramids, and Shift Scheduling at this point and should be working on Riding Circuit.
  - Have questions? Stop by the LaIR, email your section leader, or visit Piazza!
- Assignment 4 will go out on Wednesday.
  - We'll be holding YEAH Hours for this assignment this Wednesday at 7:00PM in room 380-380Y.

big-Onward!

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[7]	London
4:36.8		Maria Gommers	Netherlands	14 June 1969[7]	Leicester
4:35.3		Ellen Tittel	West Germany	20 August 1971[7]	Sittard
4:29.5		Paola Pigni	Italy	8 August 1973[7]	Viareggio
4:23.8		Natalia Mărășescu	Romania	21 May 1977[7]	Bucharest
4:22.1	4:22.09	Natalia Mărășescu	Romania	27 January 1979[7]	Auckland
4:21.7	4:21.68	Mary Decker	United States	26 January 1980[7]	Auckland
4:20.89		Lyudmila Veselkova	Soviet Union	12 September 1981[7]	Bologna
4:18.08		Mary Decker-Tabb	United States	9 July 1982[7]	Paris
4:17.44		Maricica Puică	Romania	9 September 1982[7]	Rieti
4:16.71		Mary Decker-Slaney	United States	21 August 1985[7]	Zürich
4:15.61		Paula Ivan	Romania	10 July 1989[7]	Nice
4:12.56		Svetlana Masterkova	Russia	14 August 1996[7]	Zürich

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.

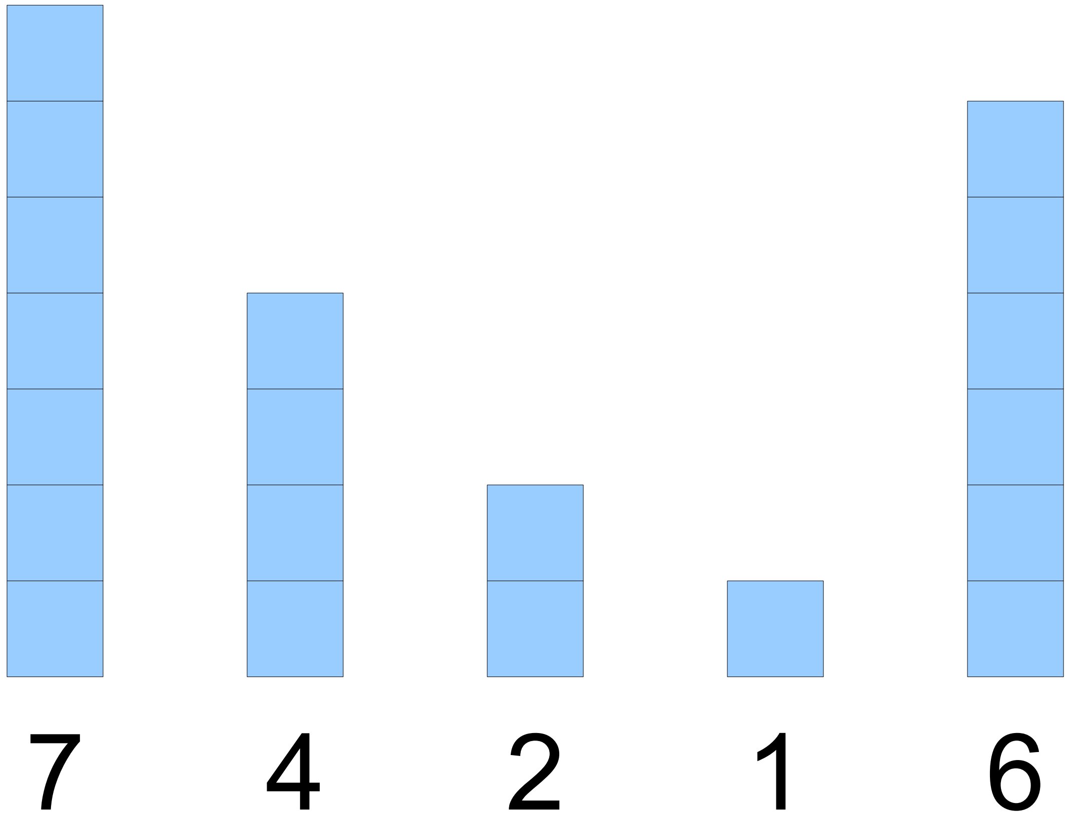
What are some strategies we could use?

How do those strategies compare?

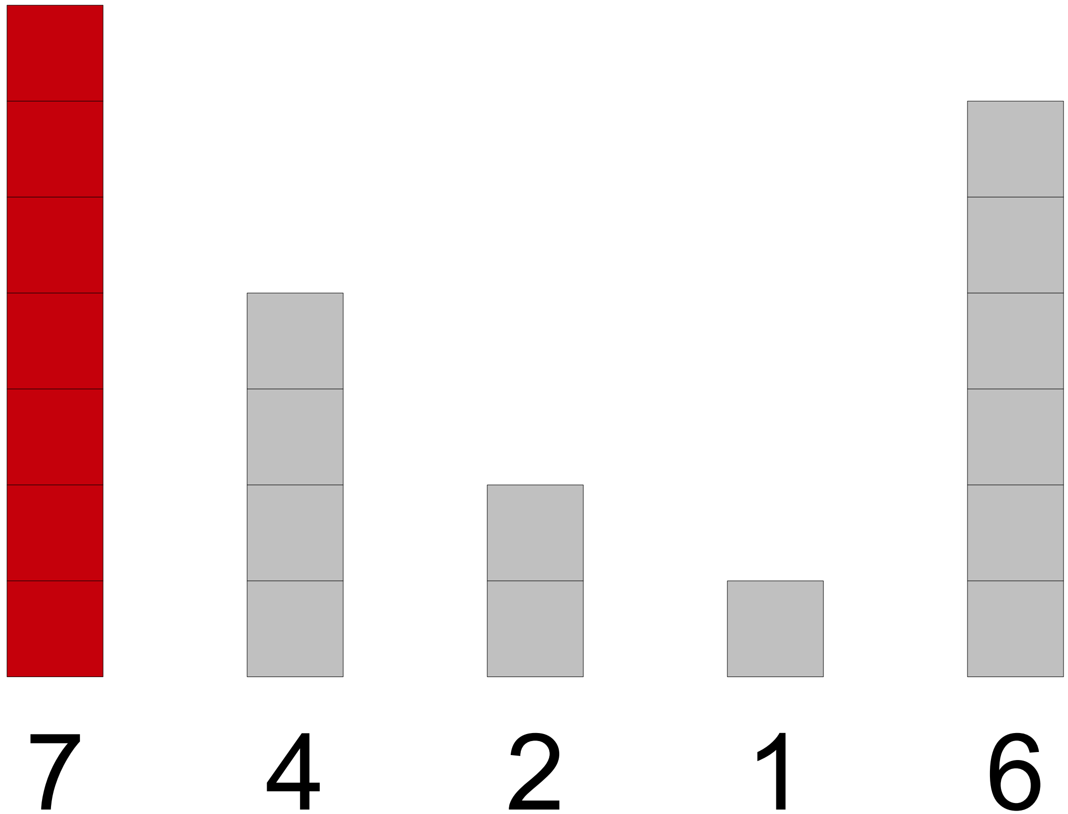
Is there a "best" strategy?

An Initial Idea: ***Insertion Sort***

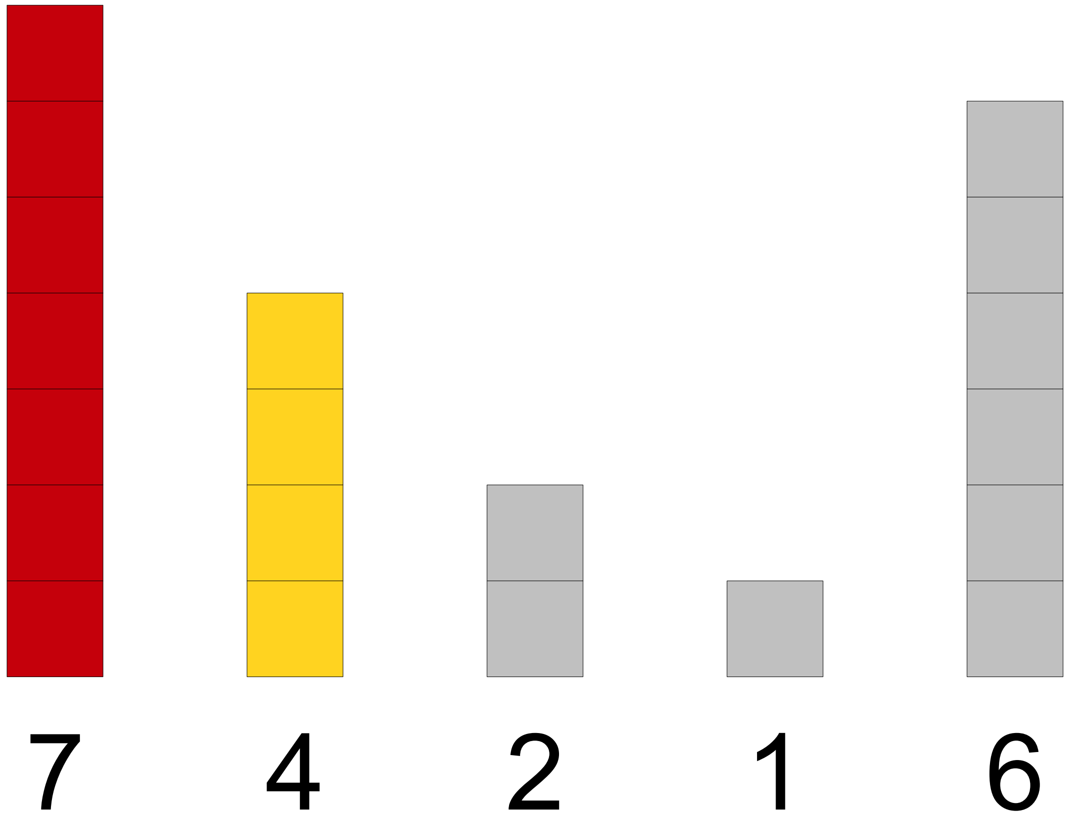
An Initial Idea: ***Insertion Sort***



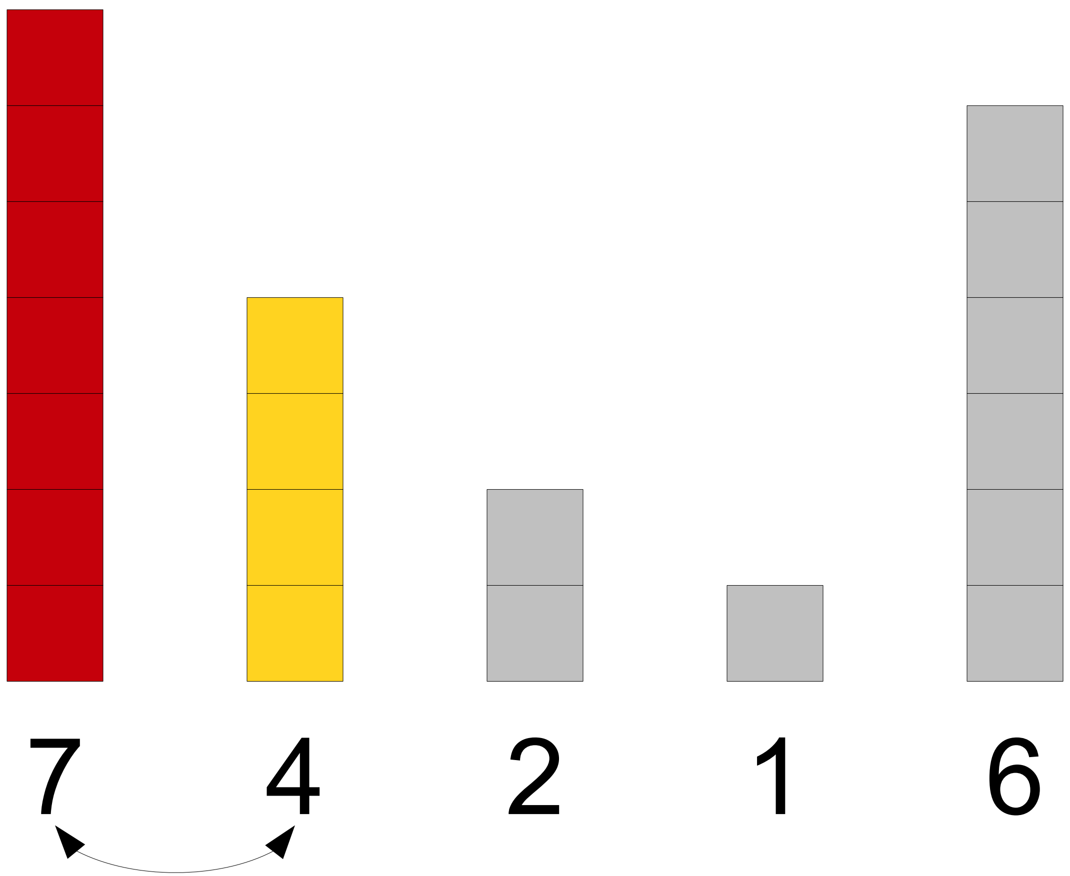
An Initial Idea: ***Insertion Sort***



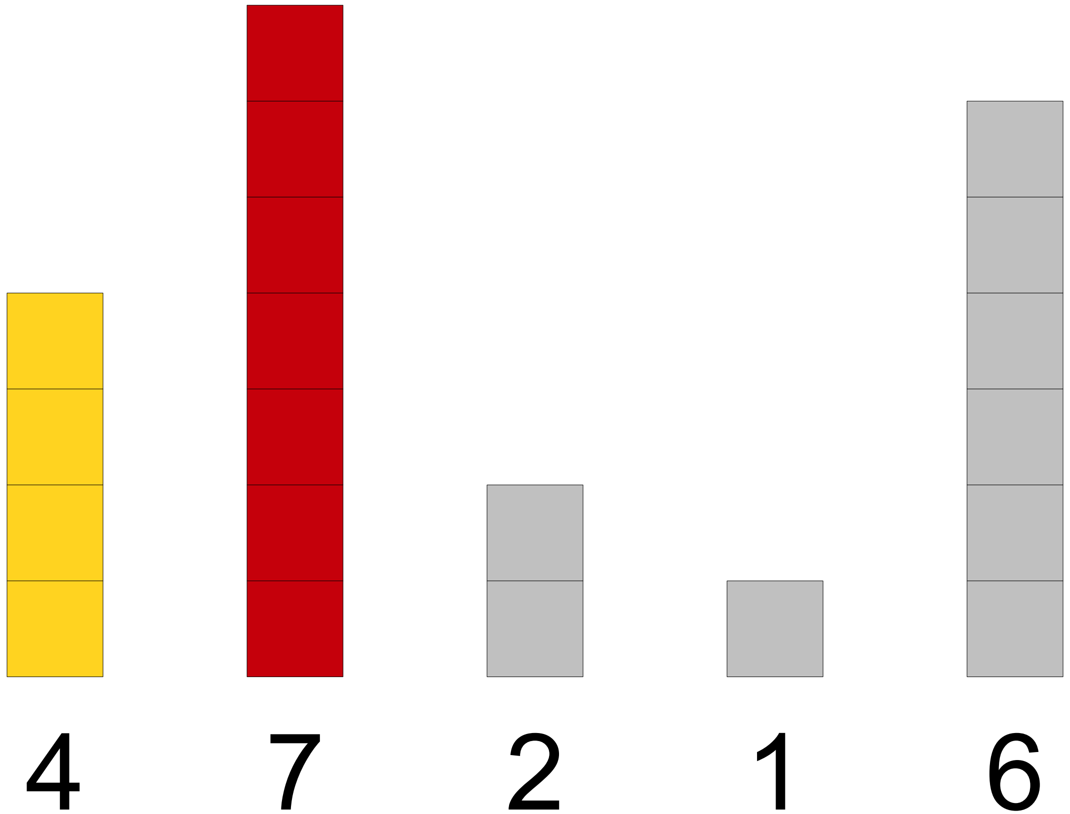
An Initial Idea: ***Insertion Sort***



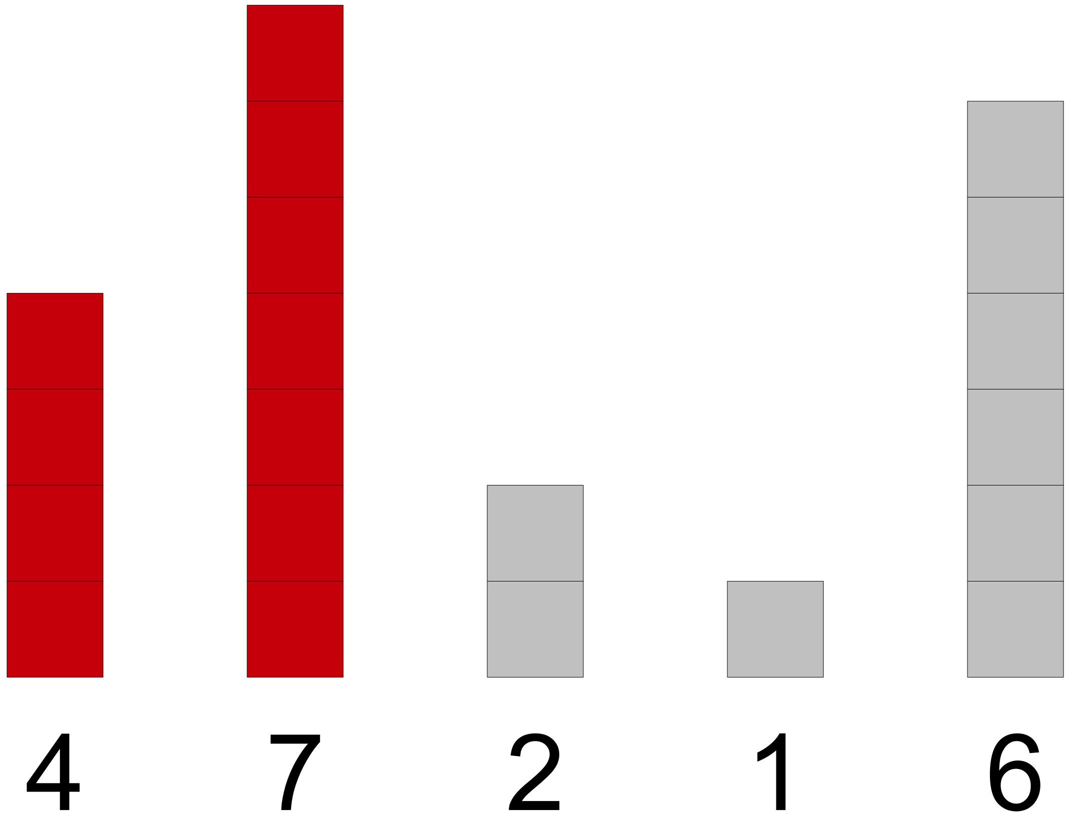
An Initial Idea: ***Insertion Sort***



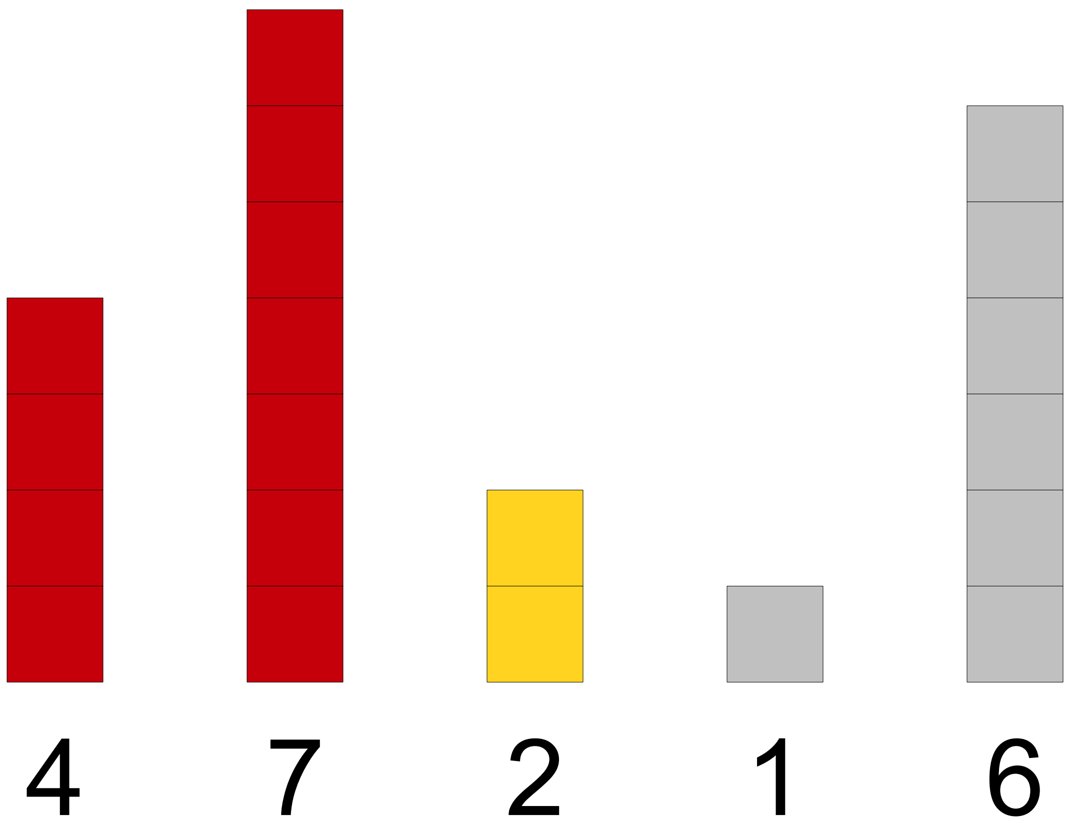
An Initial Idea: ***Insertion Sort***



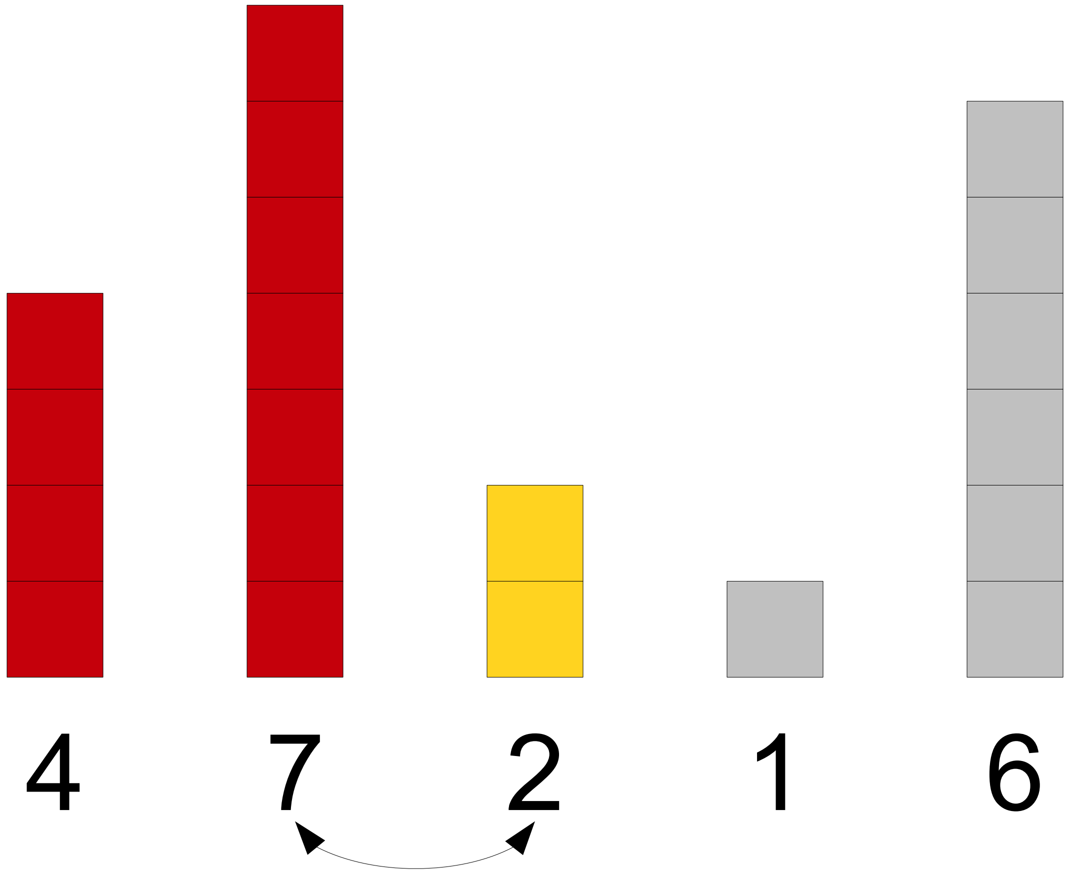
An Initial Idea: ***Insertion Sort***



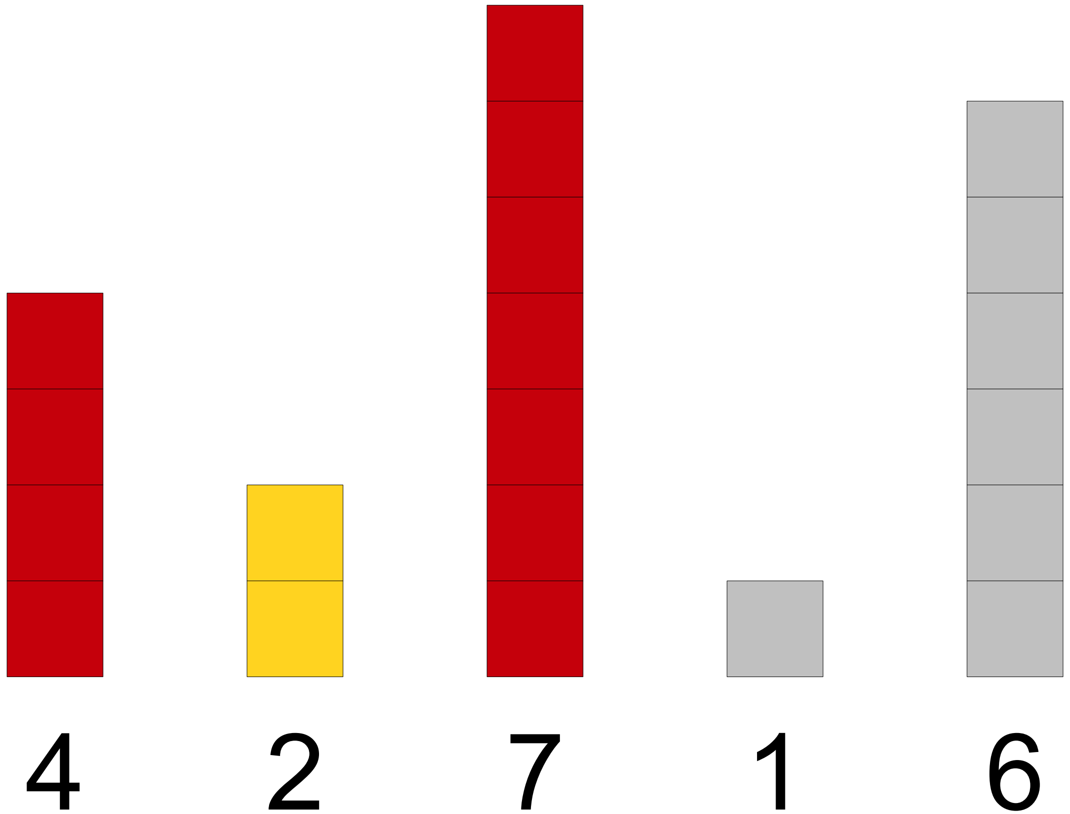
An Initial Idea: ***Insertion Sort***



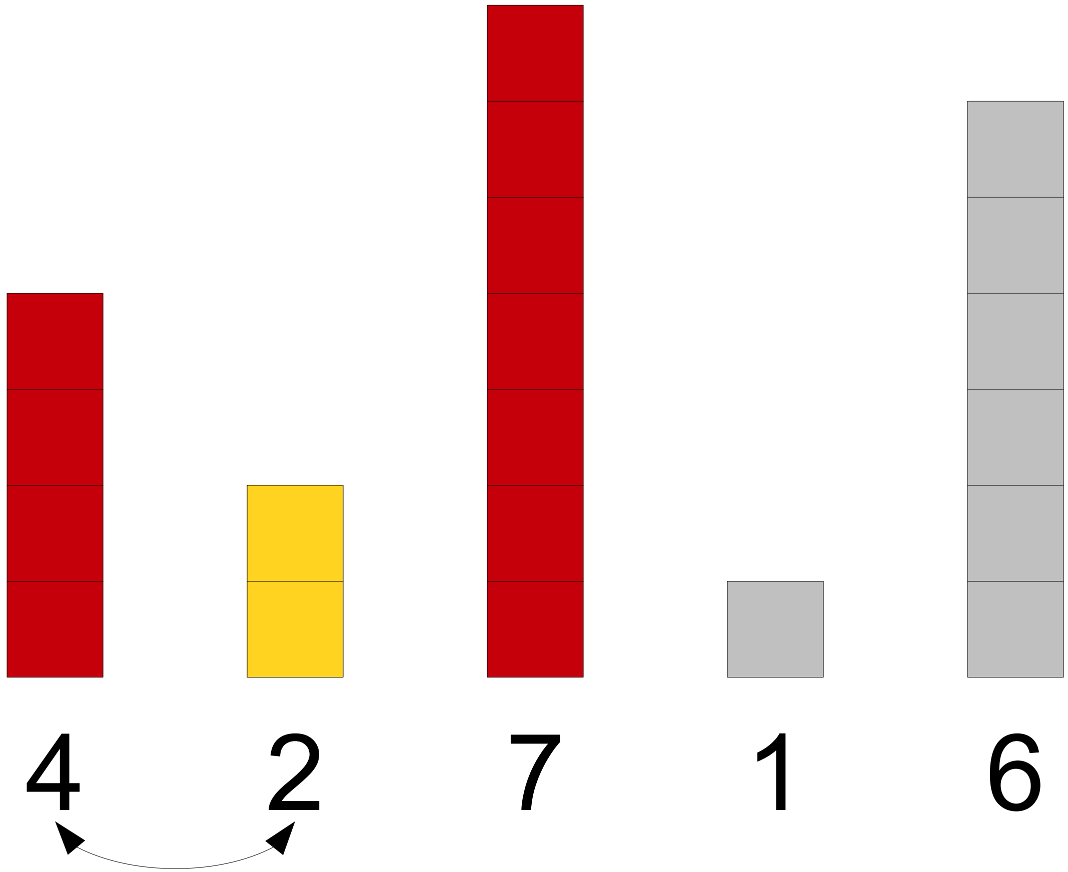
An Initial Idea: ***Insertion Sort***



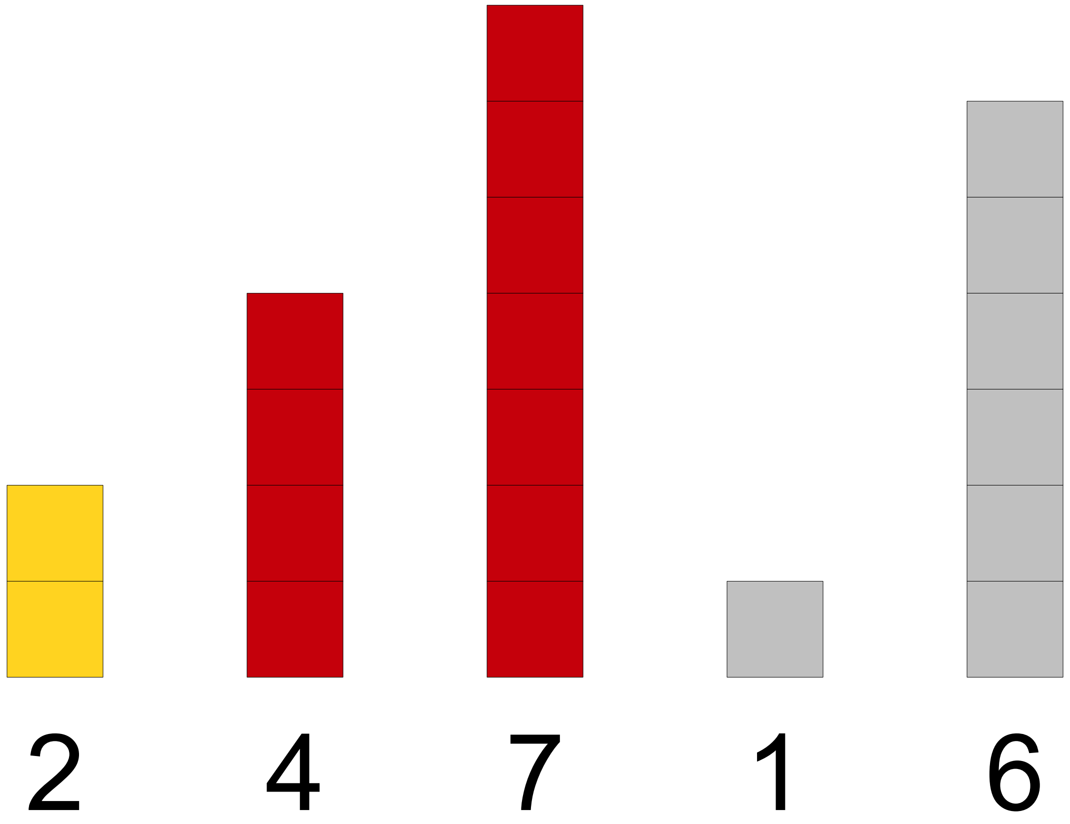
An Initial Idea: ***Insertion Sort***



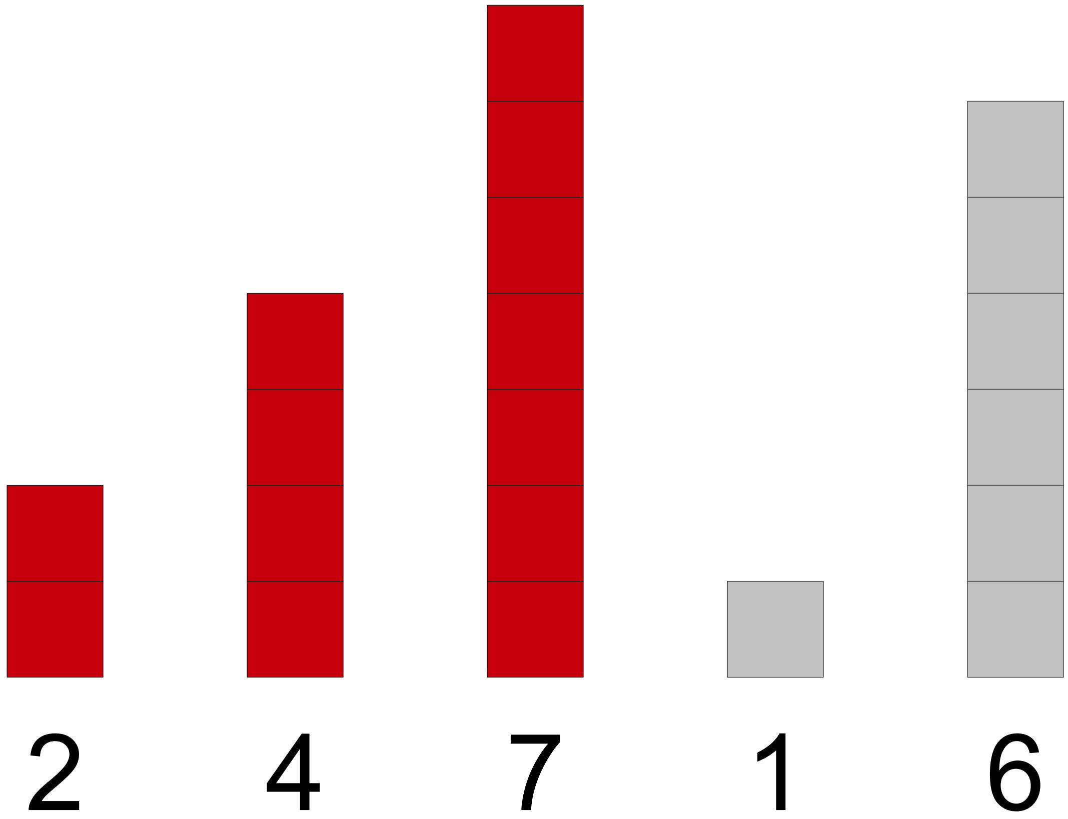
An Initial Idea: ***Insertion Sort***



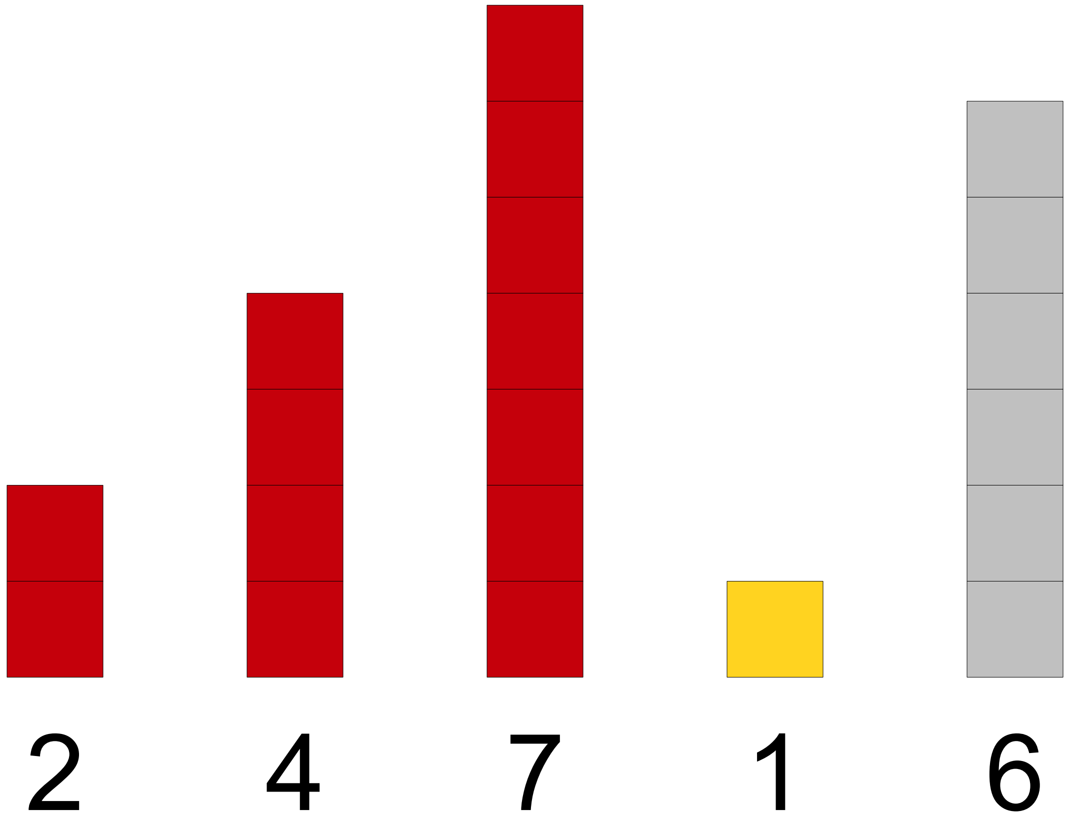
An Initial Idea: ***Insertion Sort***



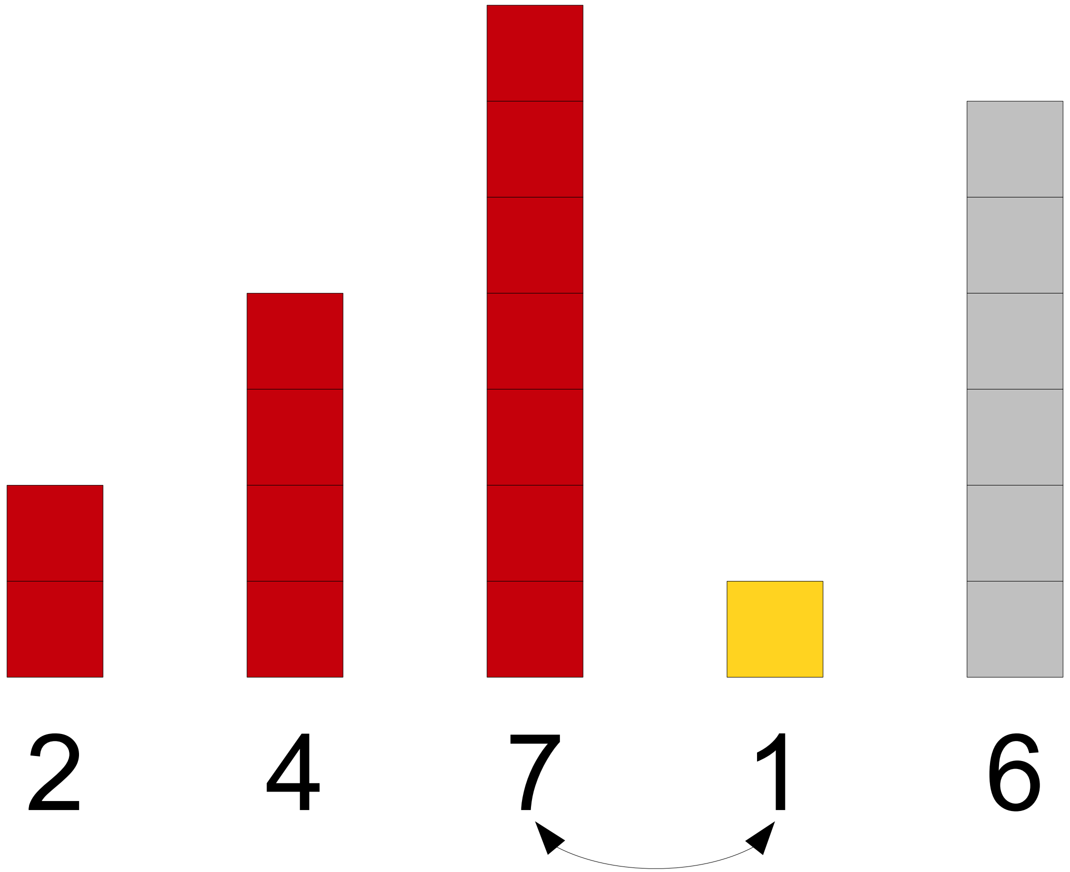
An Initial Idea: ***Insertion Sort***



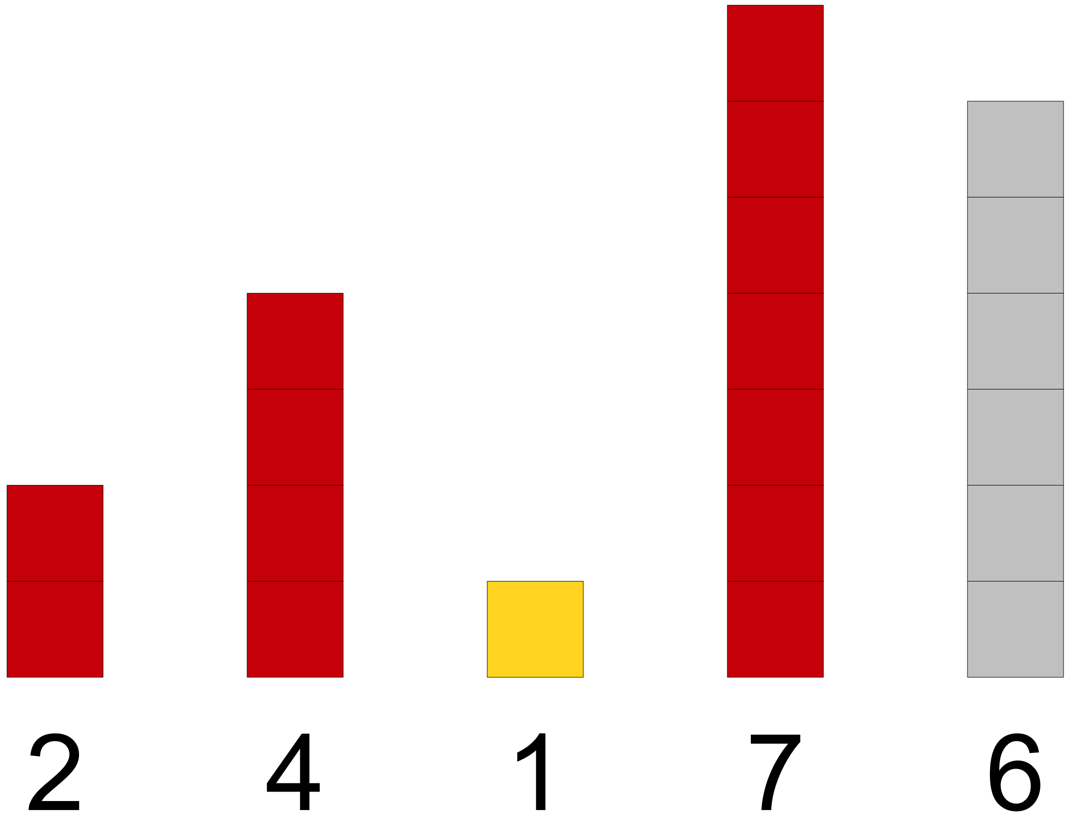
An Initial Idea: ***Insertion Sort***



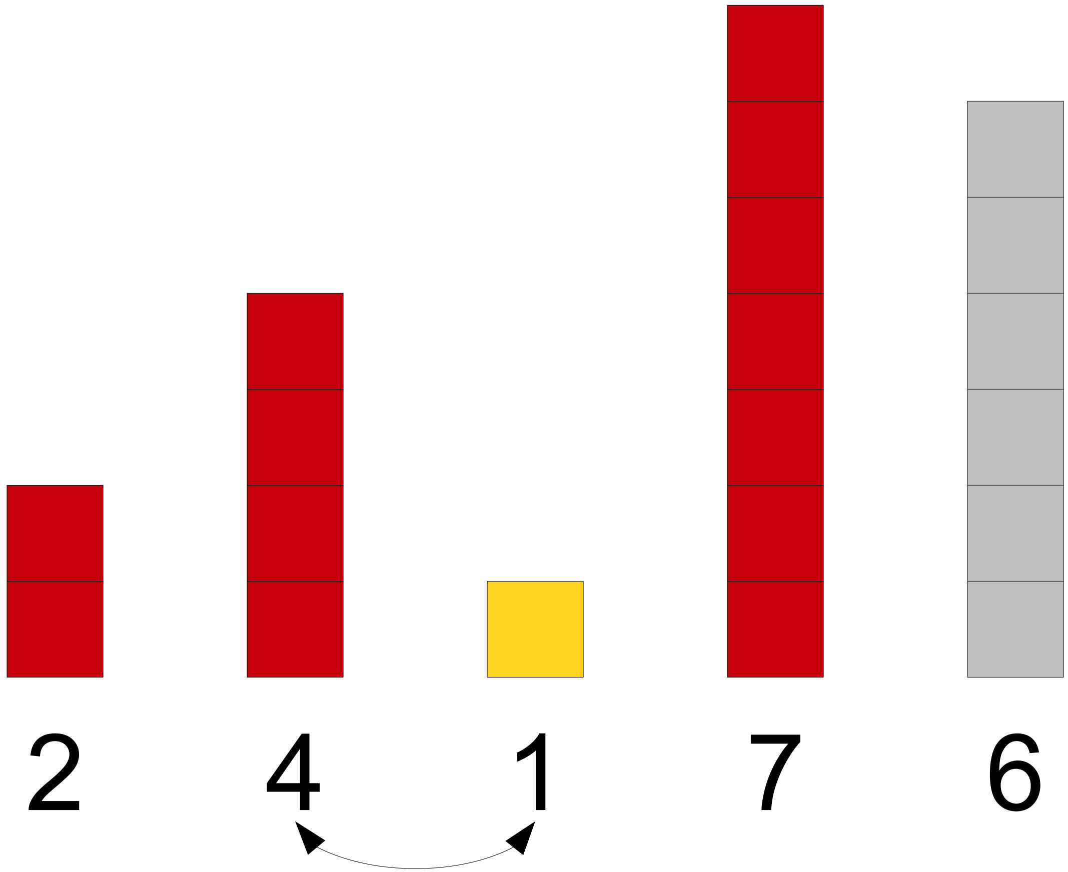
An Initial Idea: ***Insertion Sort***



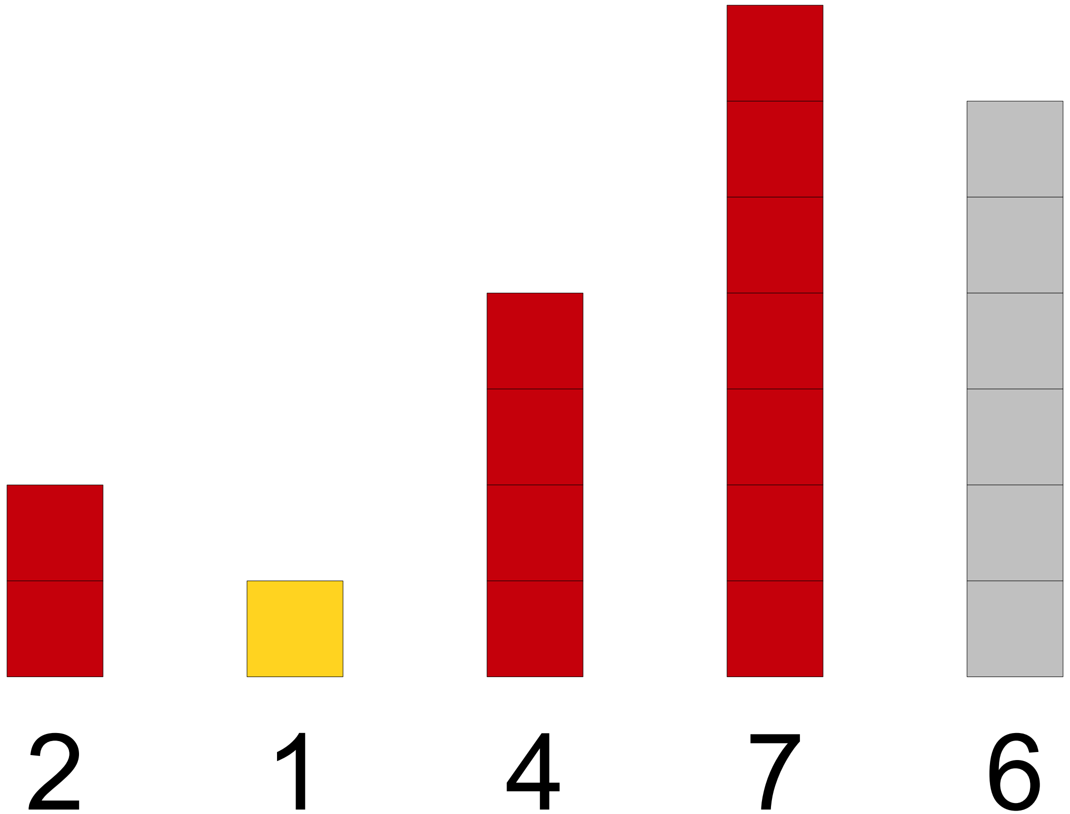
An Initial Idea: ***Insertion Sort***



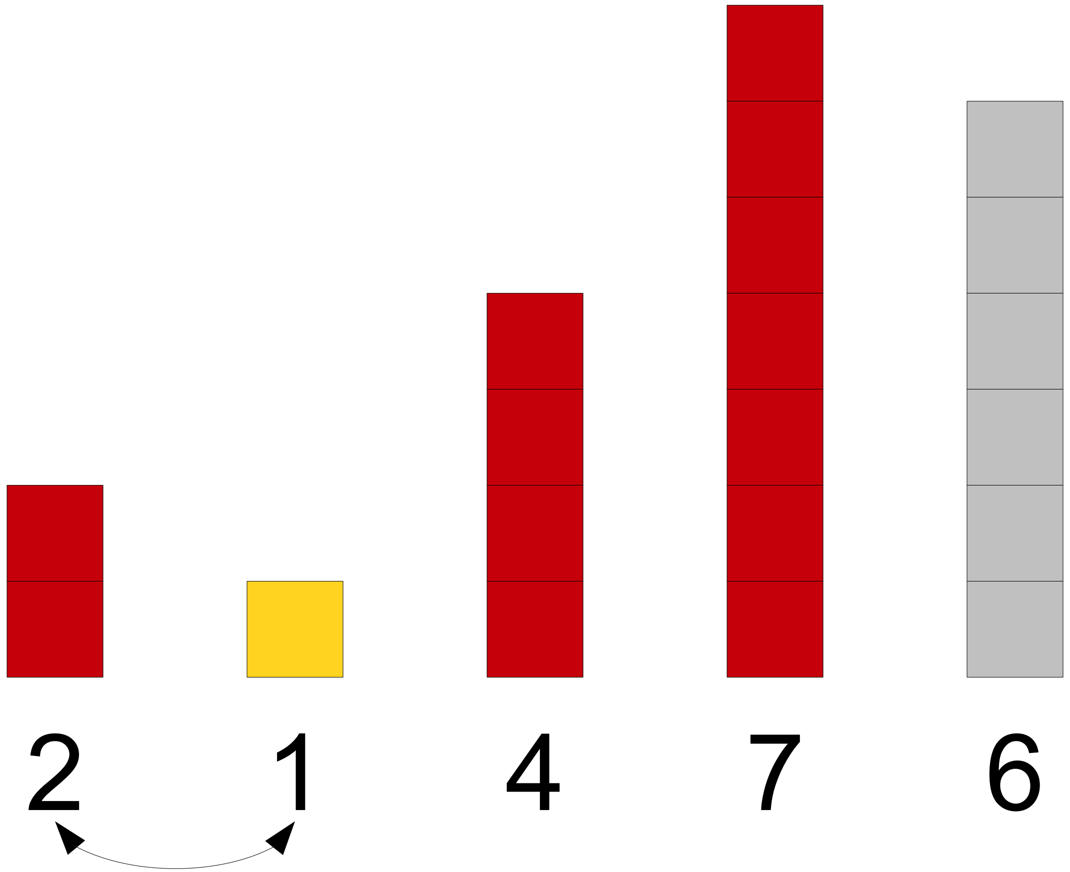
An Initial Idea: ***Insertion Sort***



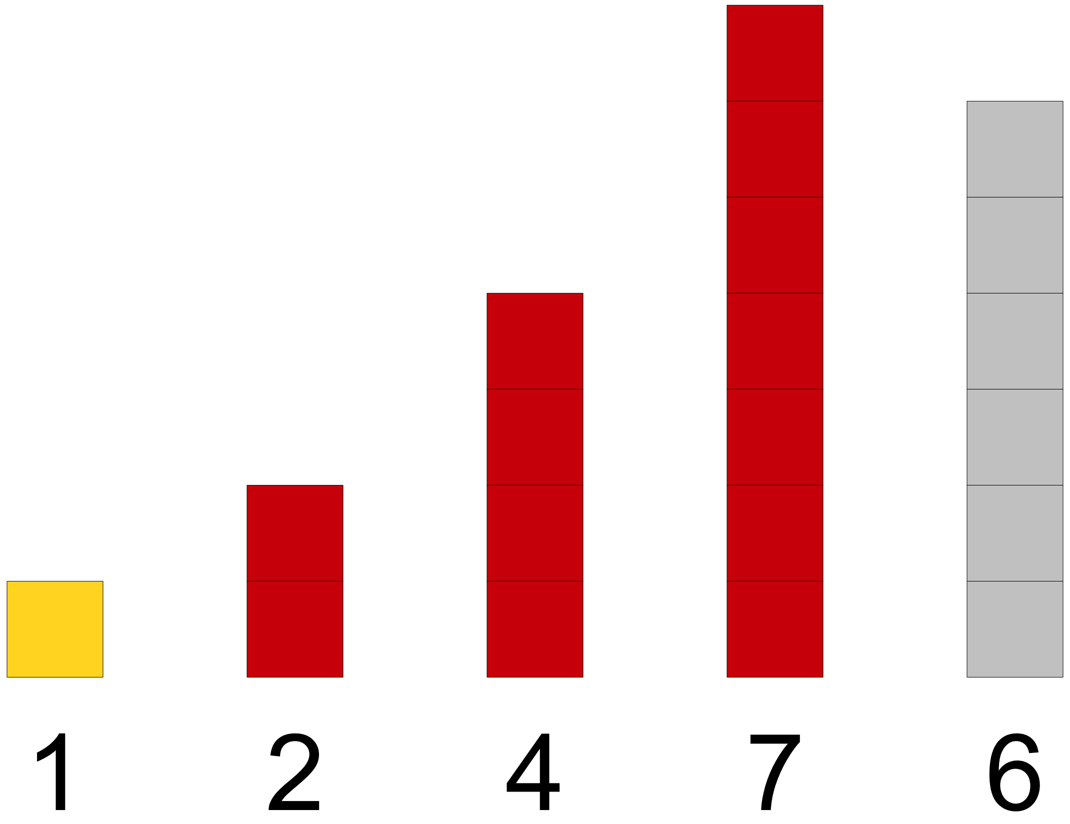
An Initial Idea: ***Insertion Sort***



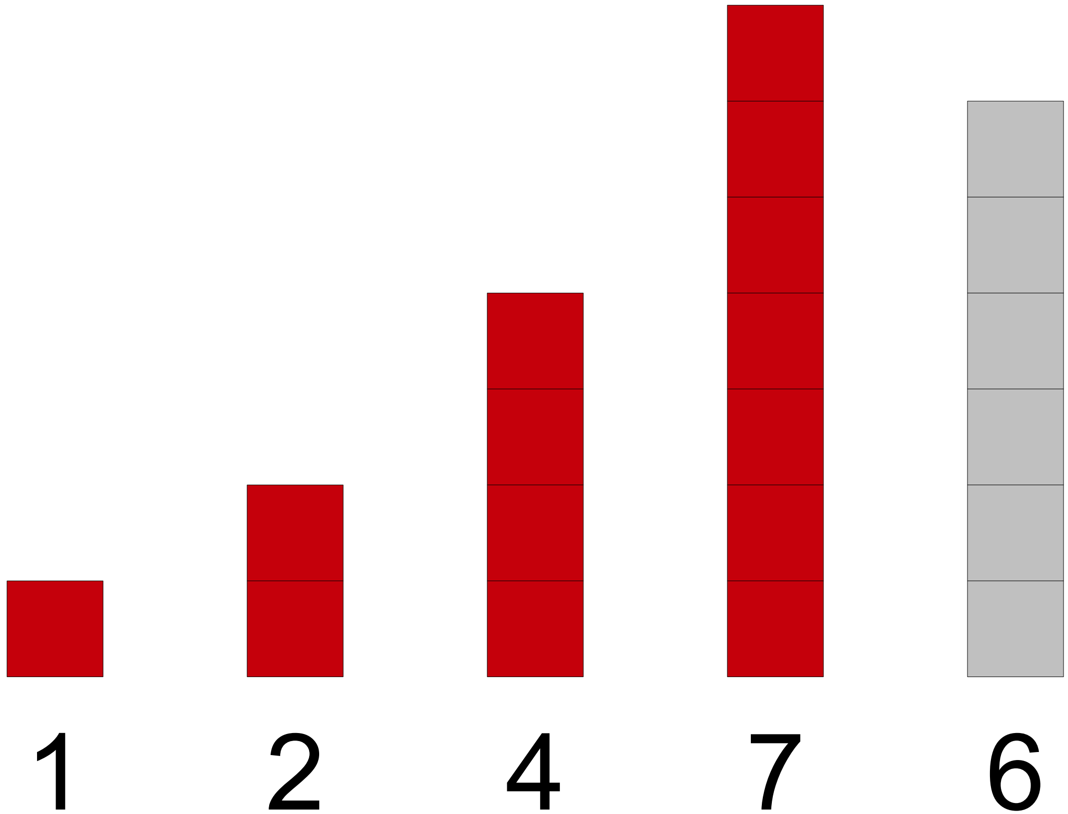
An Initial Idea: ***Insertion Sort***



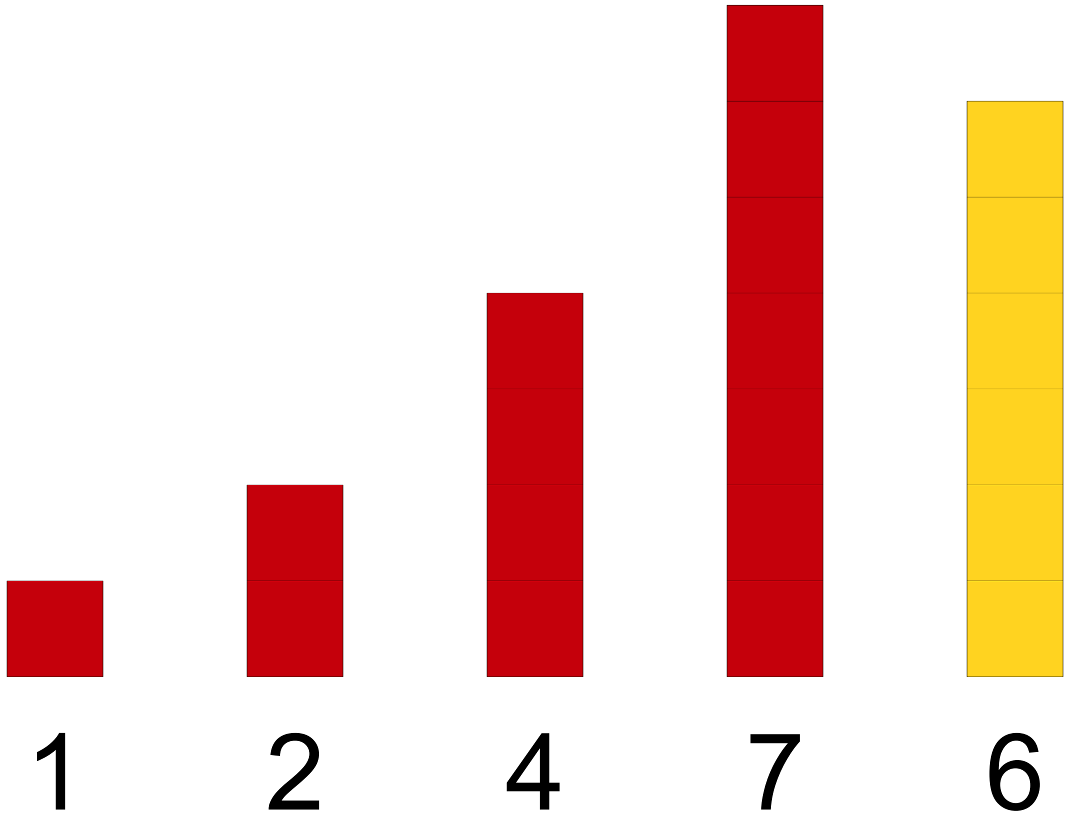
An Initial Idea: ***Insertion Sort***



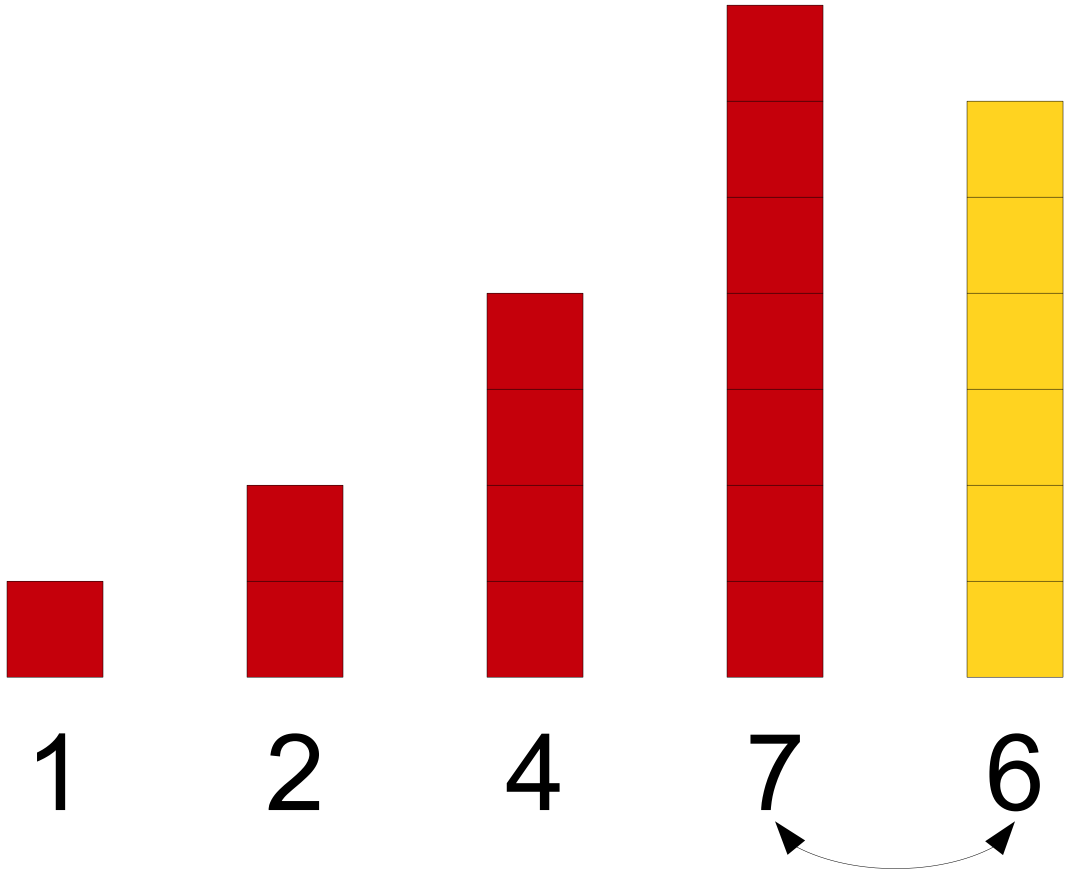
An Initial Idea: ***Insertion Sort***



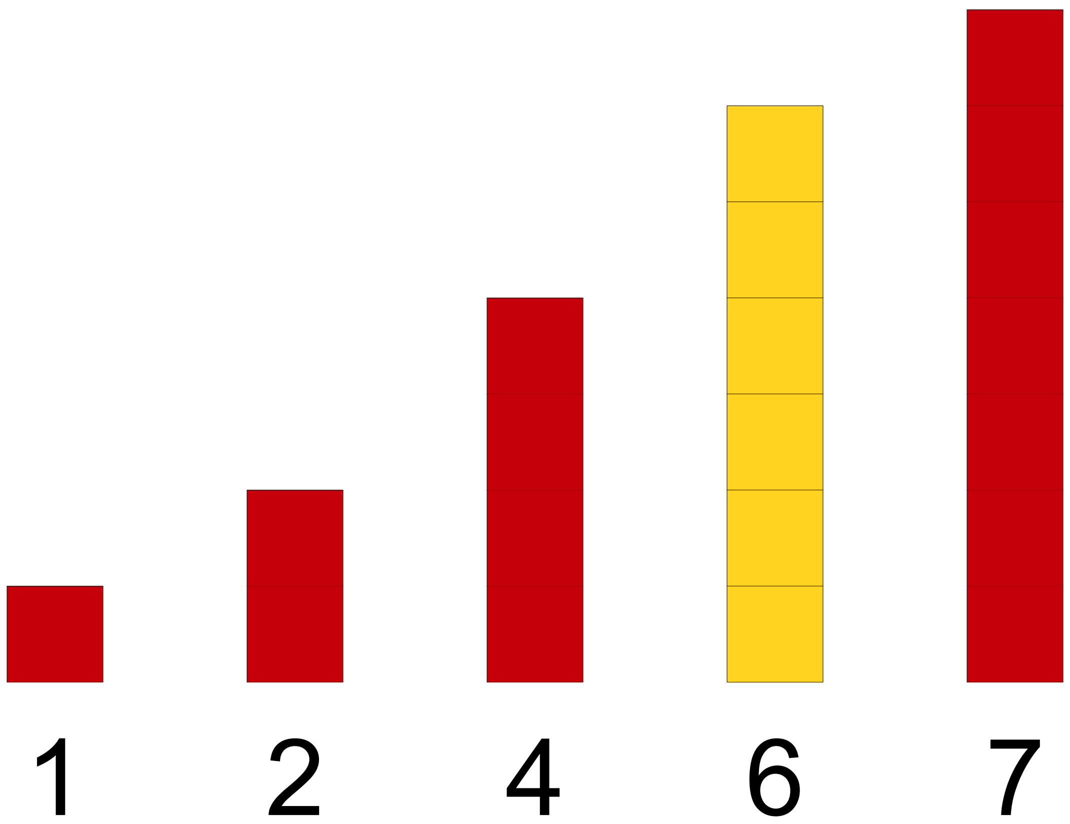
An Initial Idea: ***Insertion Sort***



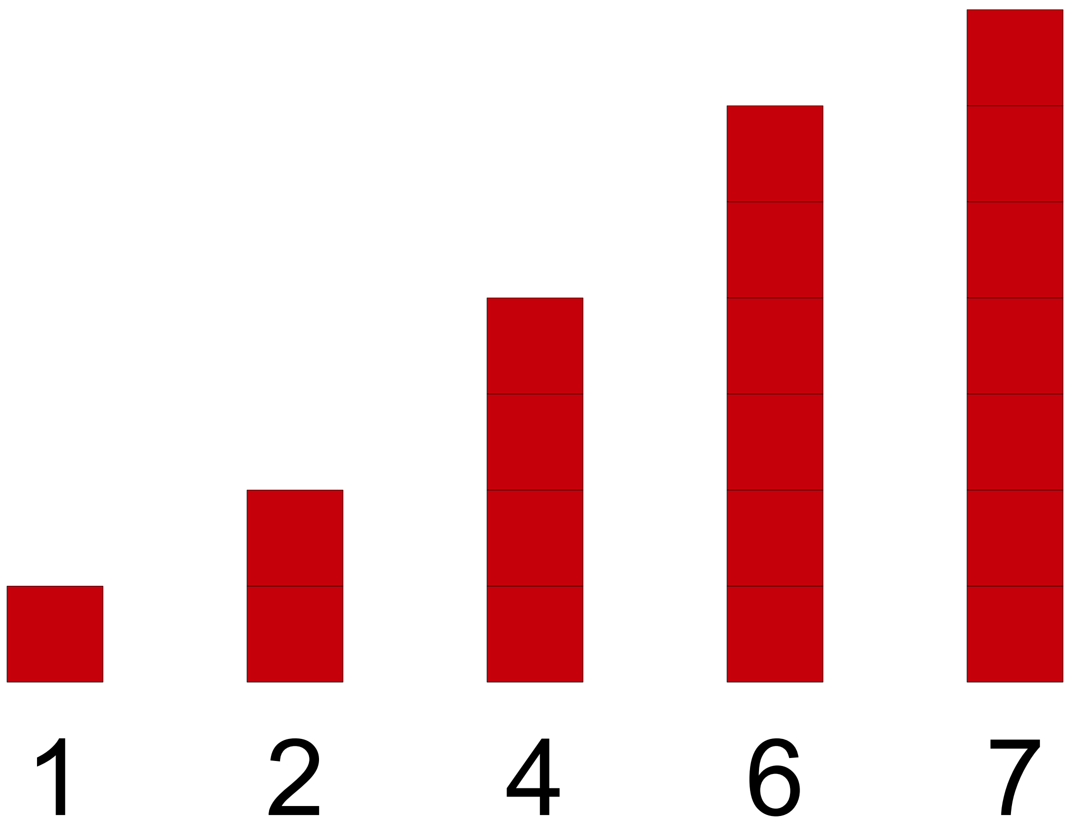
An Initial Idea: ***Insertion Sort***



An Initial Idea: ***Insertion Sort***



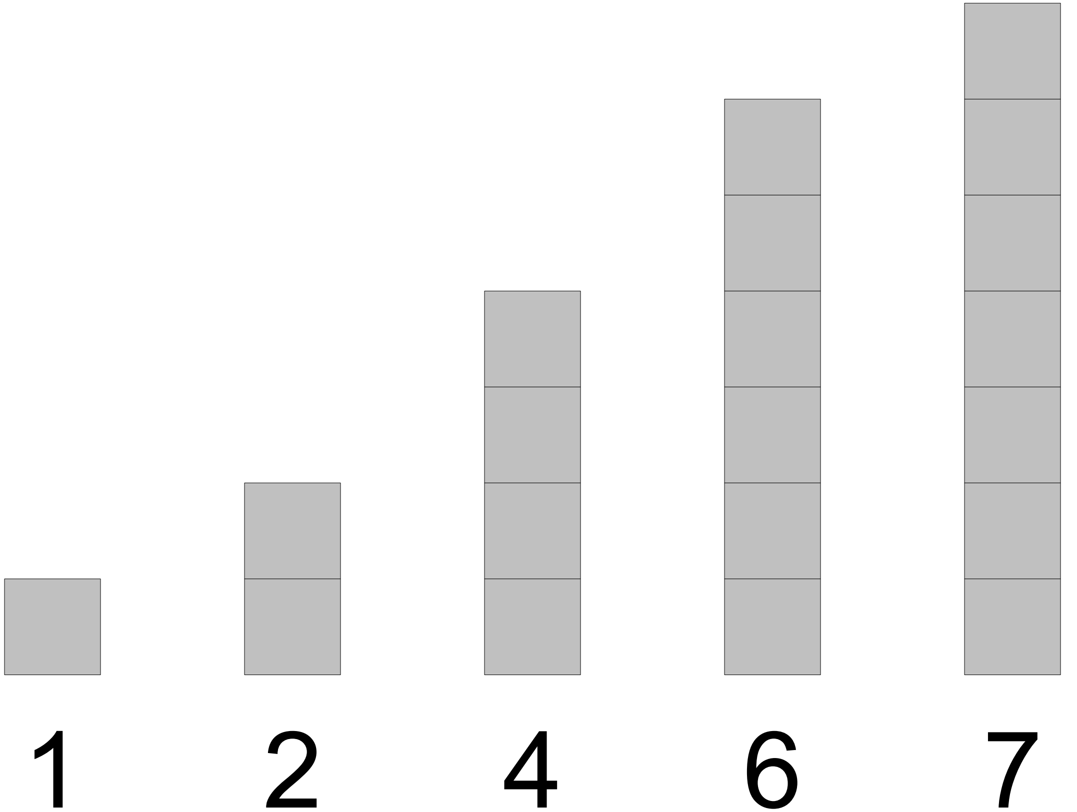
An Initial Idea: ***Insertion Sort***



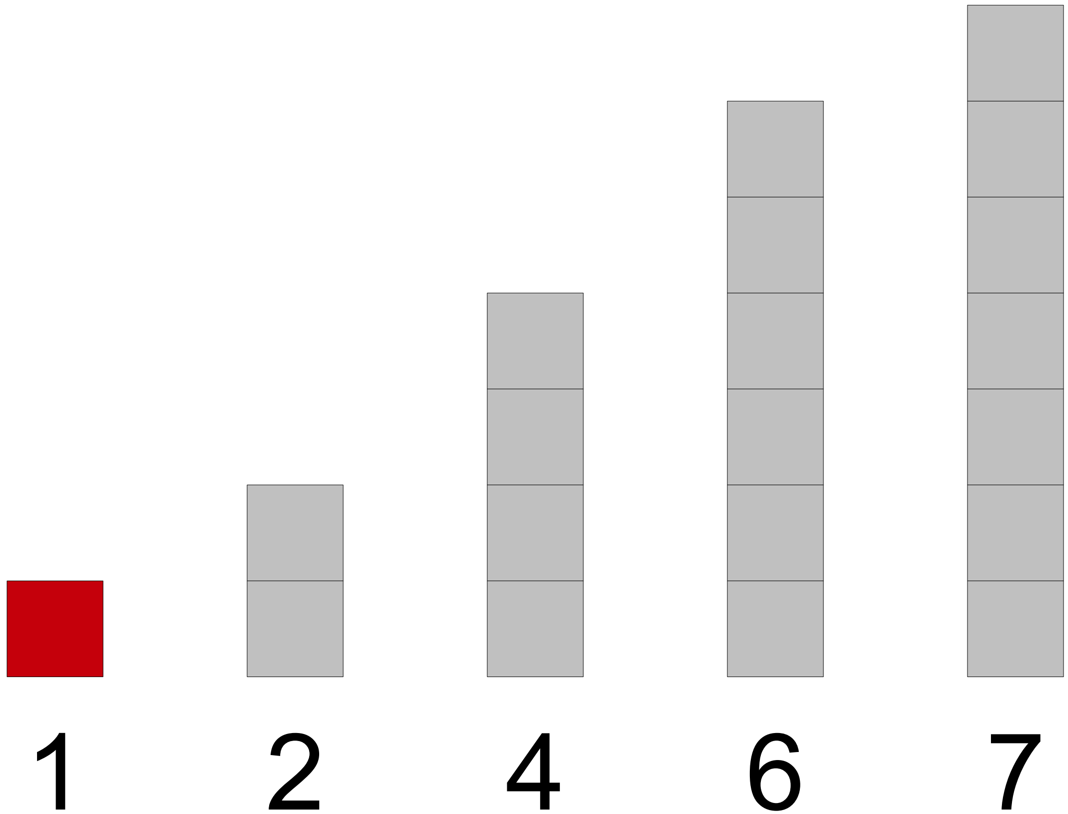
```
/**
 * Sorts the specified vector using insertion sort.
 *
 * @param v The vector to sort.
 */
void insertionSort(Vector<int>& v) {
  for (int i = 0; i < v.size(); i++) {
    /* Scan backwards until either (1) there is no
     * preceding element or the preceding element is
     * no bigger than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

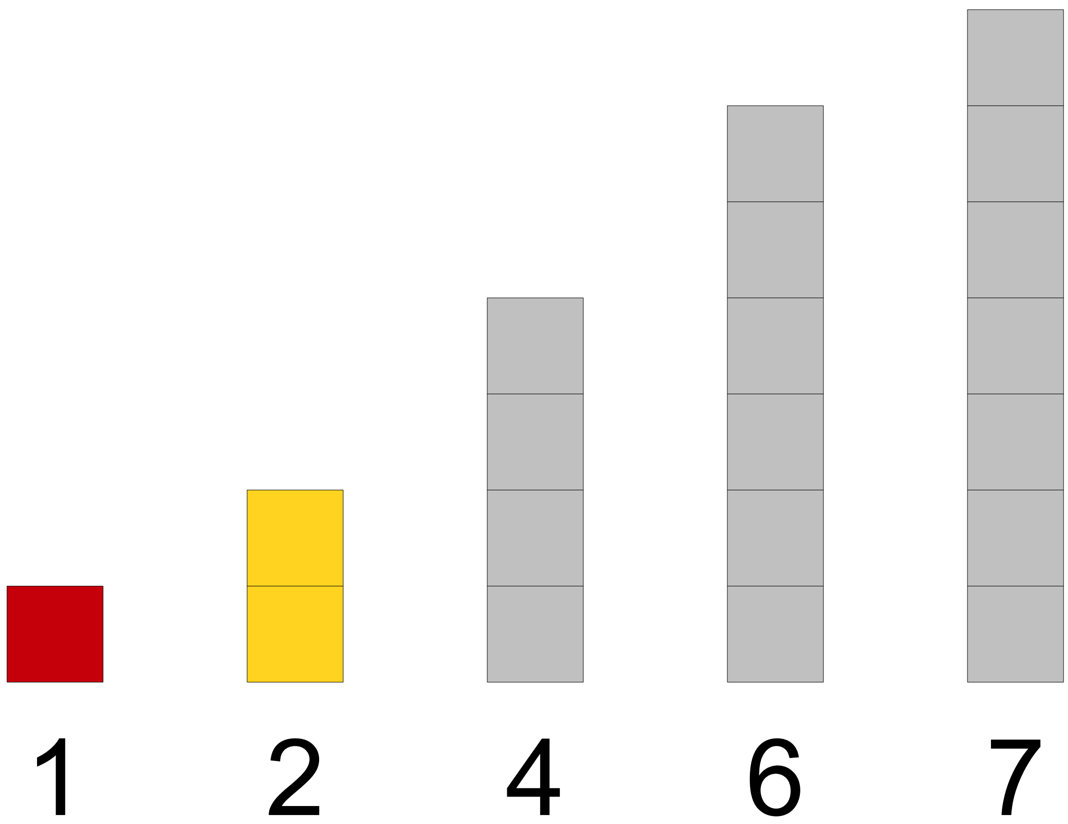
How Fast is Insertion Sort?



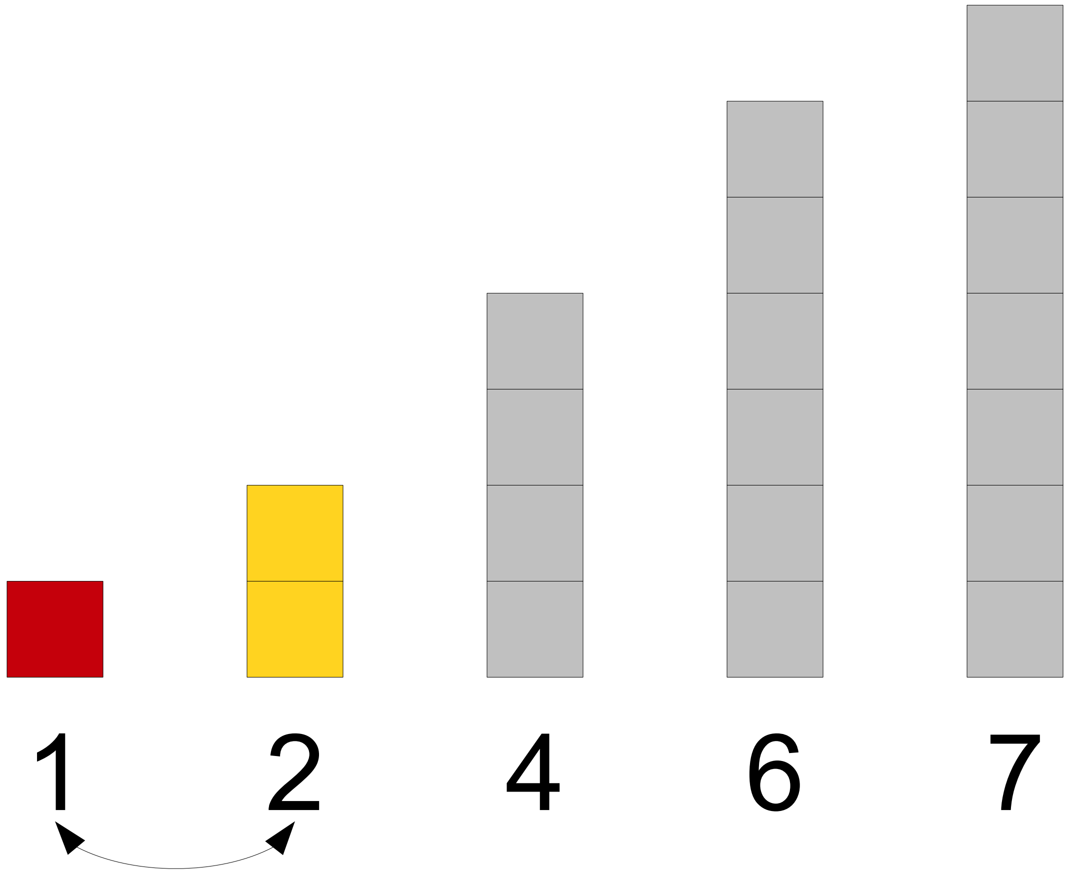
How Fast is Insertion Sort?



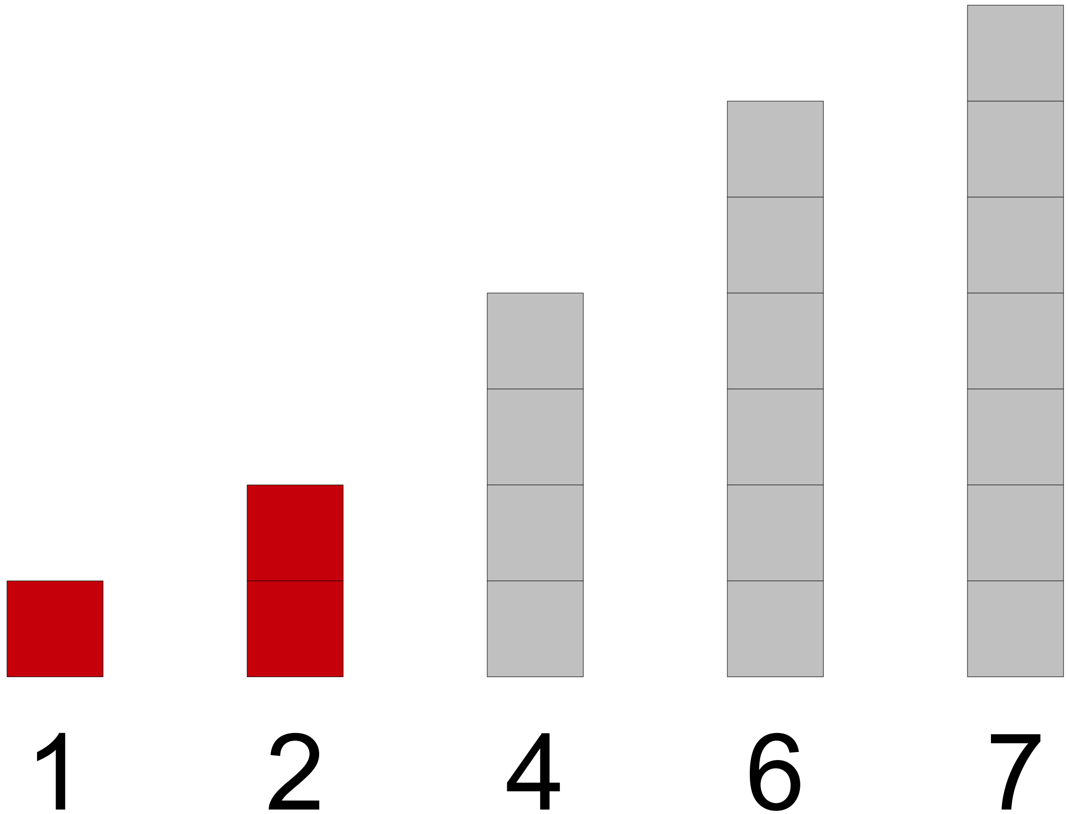
How Fast is Insertion Sort?



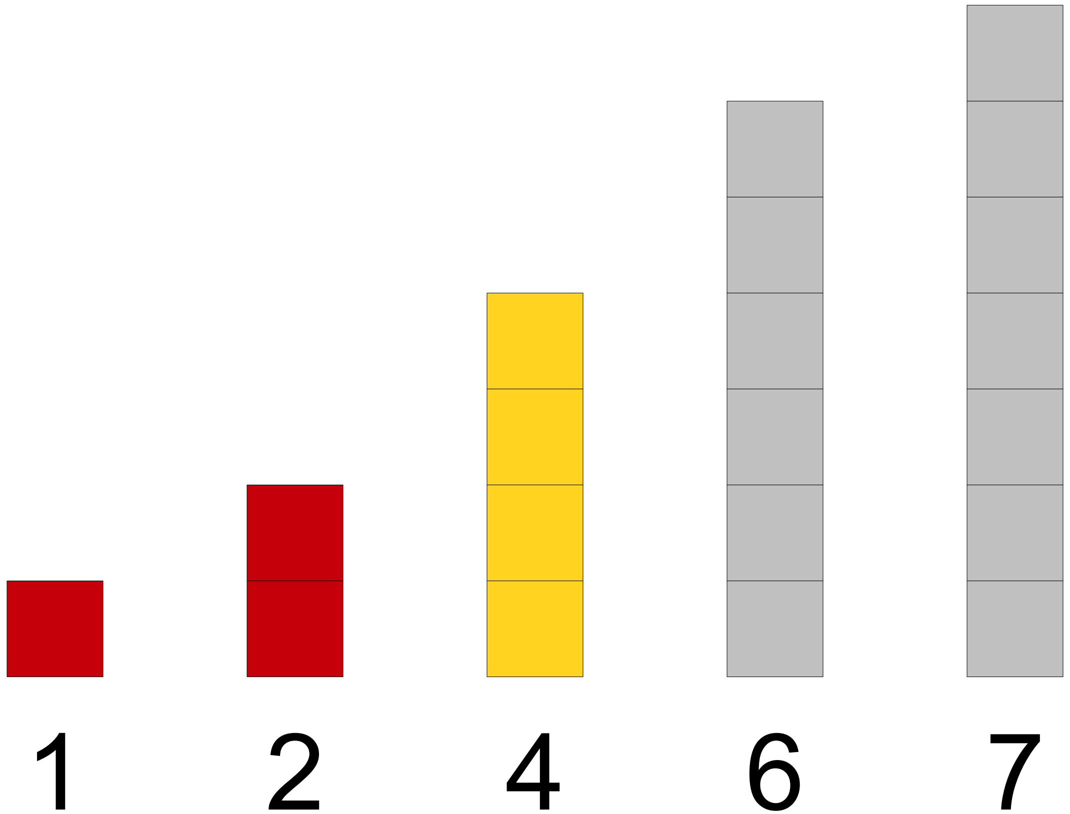
How Fast is Insertion Sort?



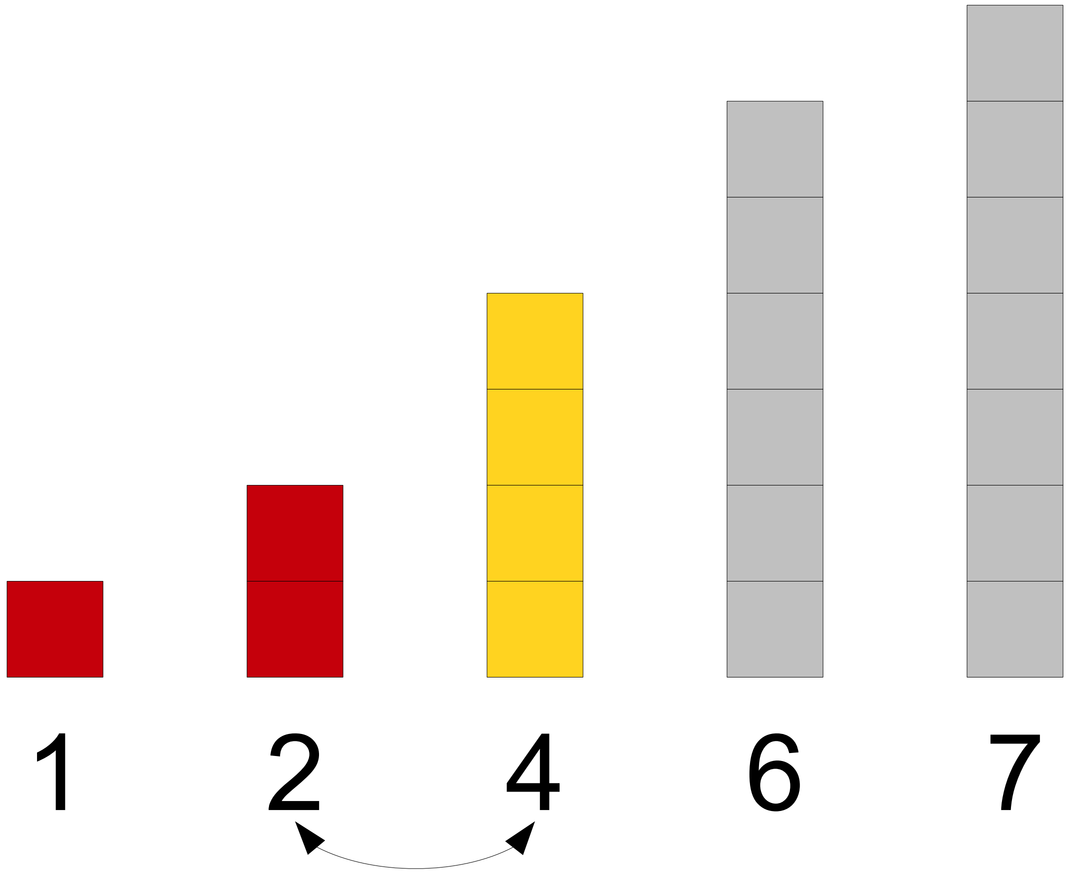
How Fast is Insertion Sort?



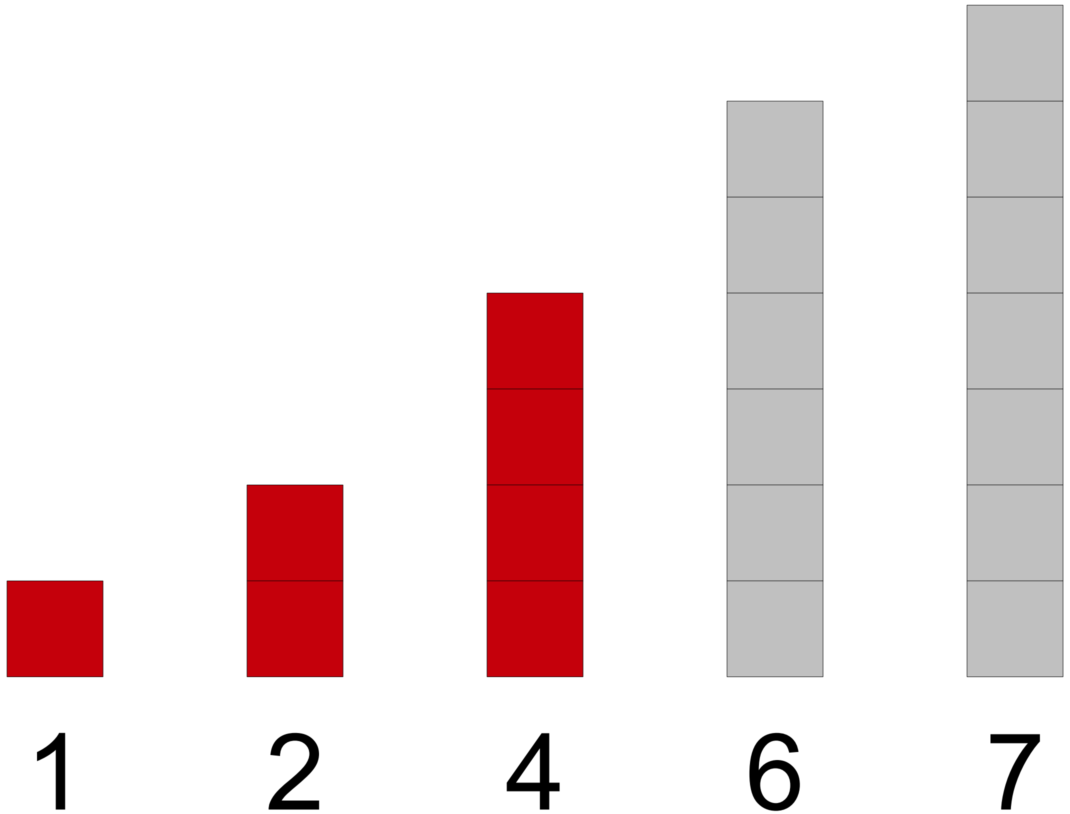
How Fast is Insertion Sort?



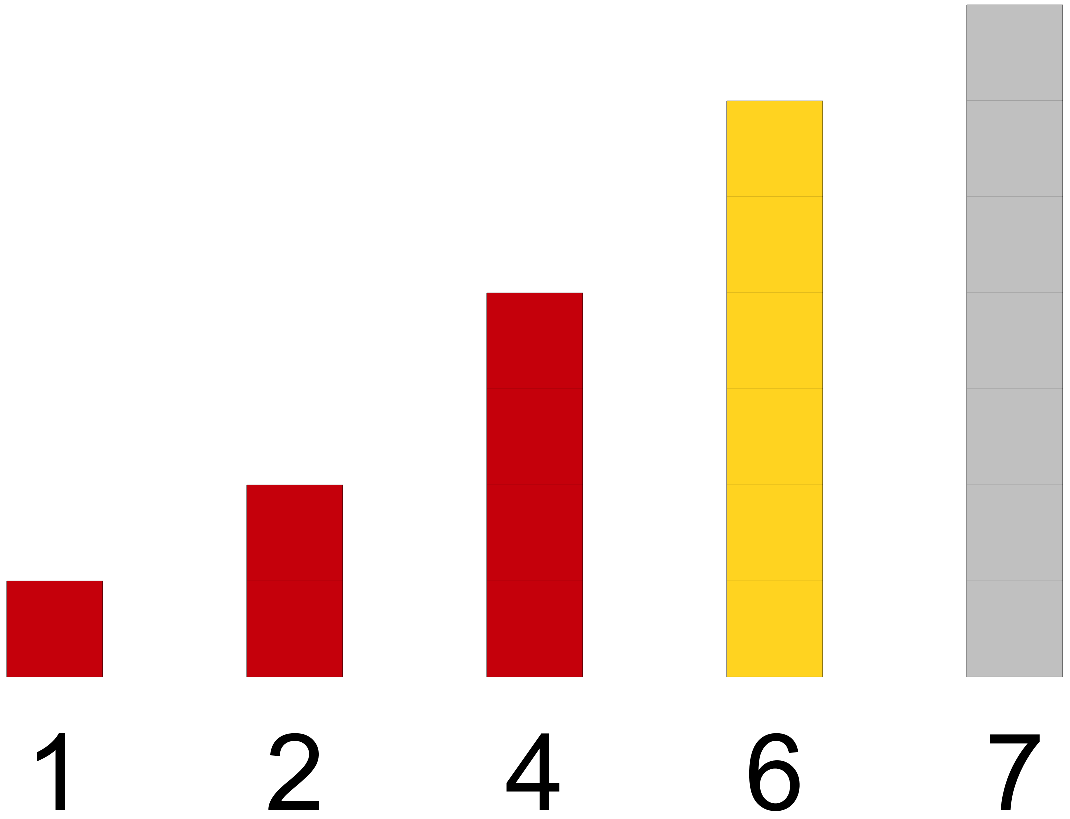
How Fast is Insertion Sort?



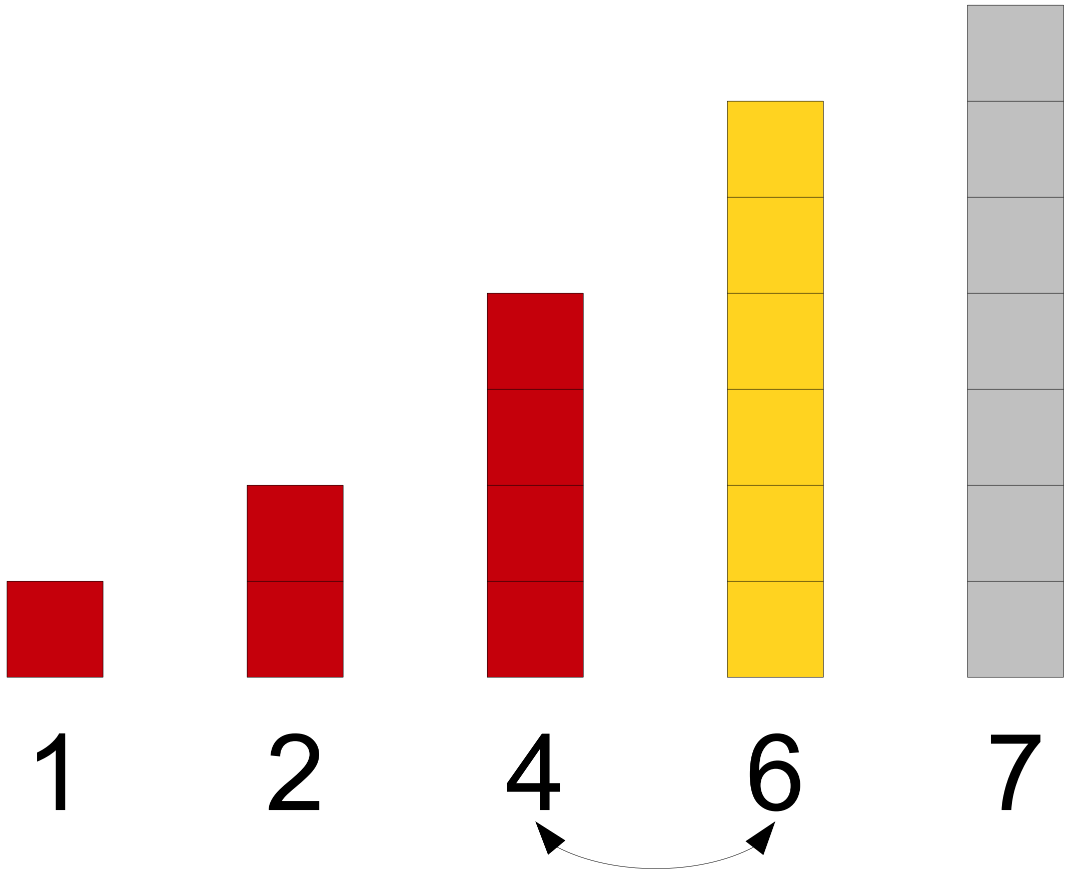
How Fast is Insertion Sort?



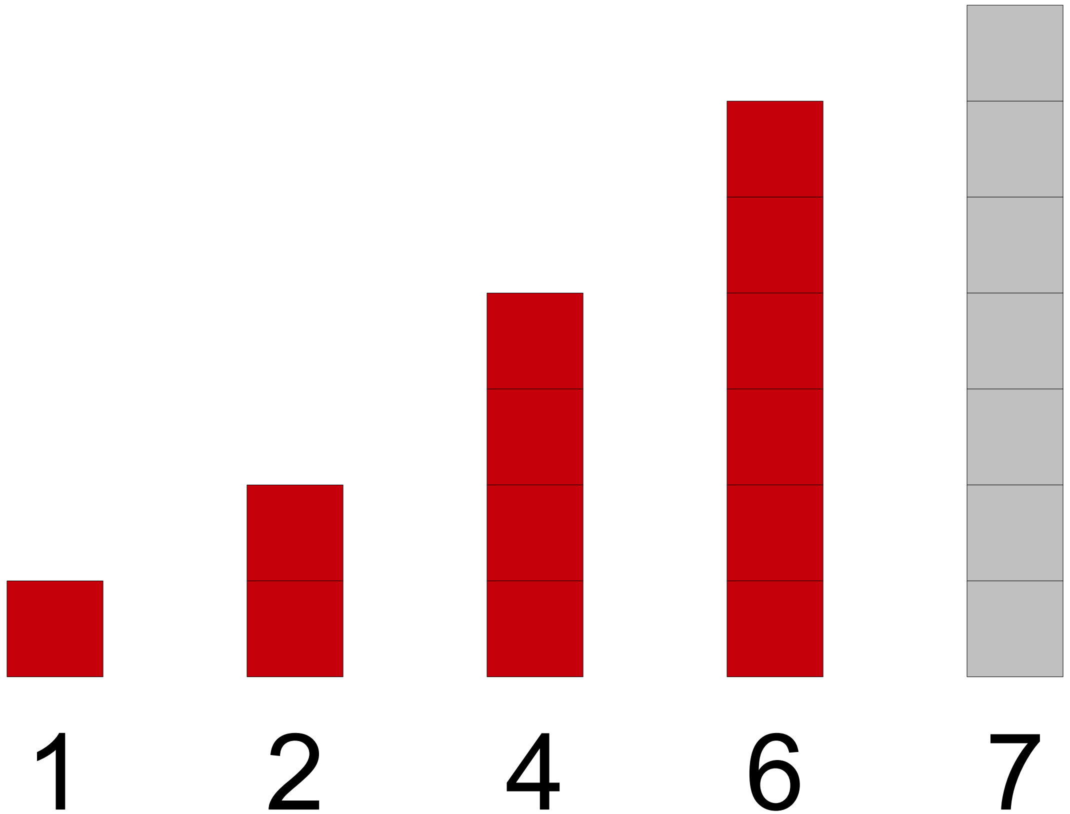
How Fast is Insertion Sort?



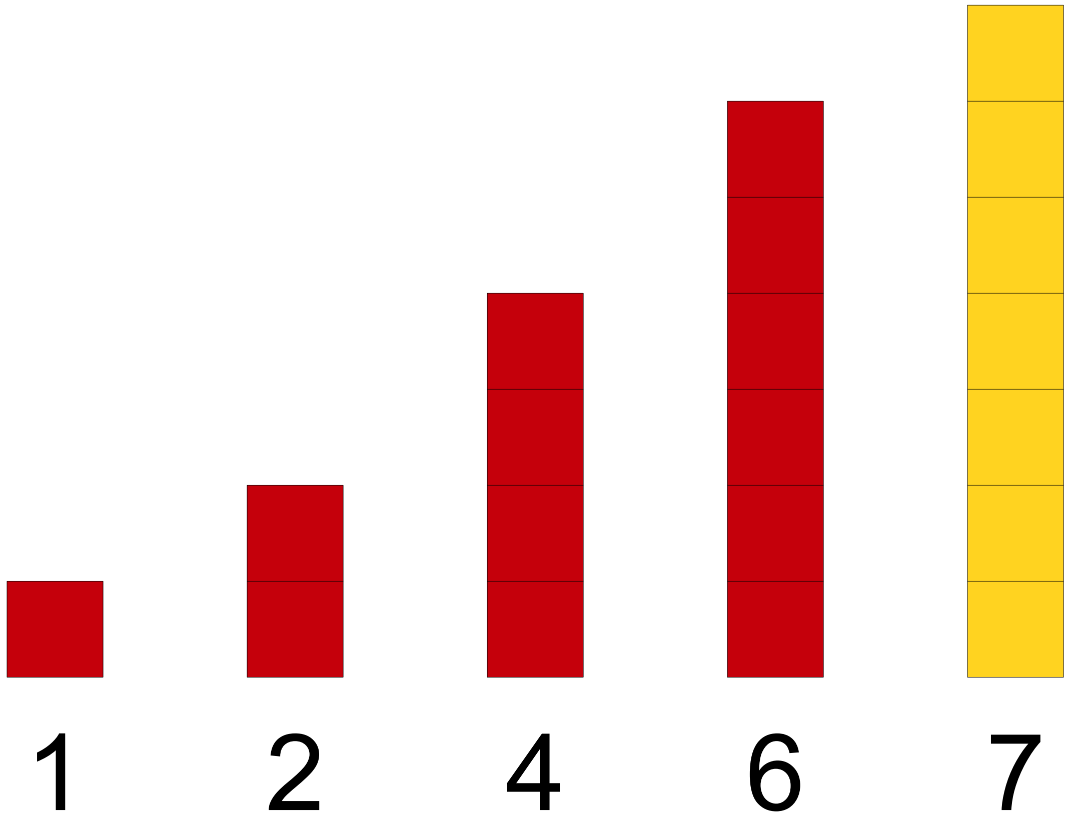
How Fast is Insertion Sort?



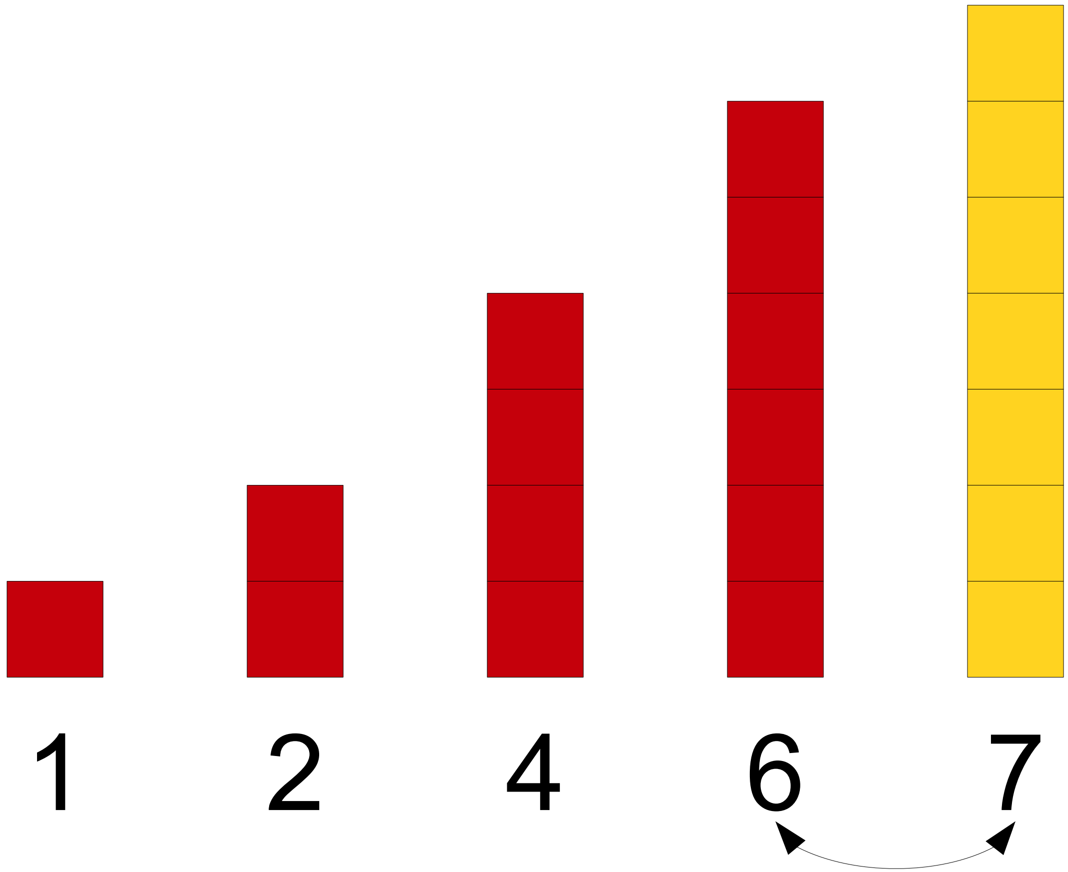
How Fast is Insertion Sort?



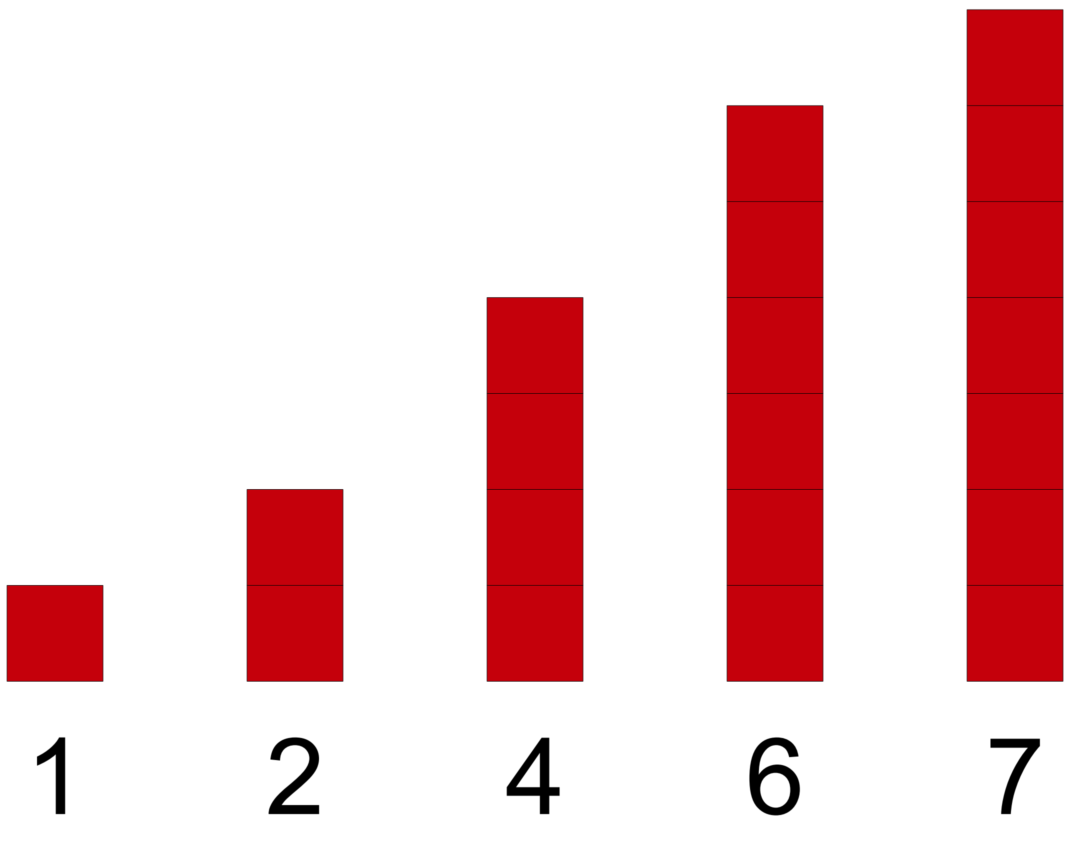
How Fast is Insertion Sort?



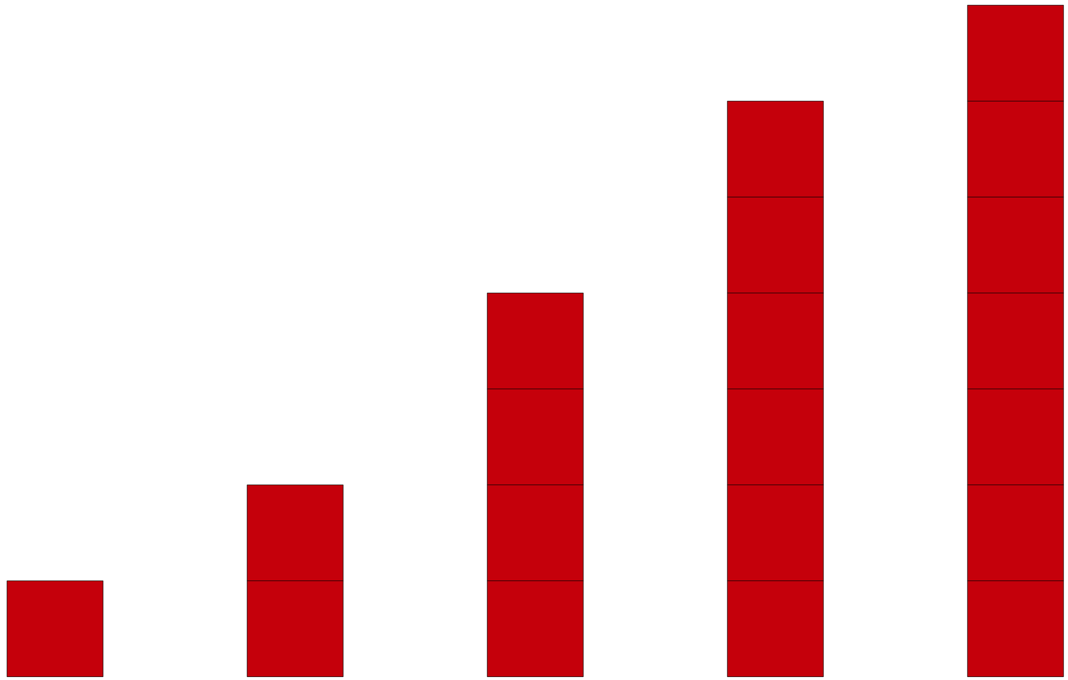
How Fast is Insertion Sort?



How Fast is Insertion Sort?



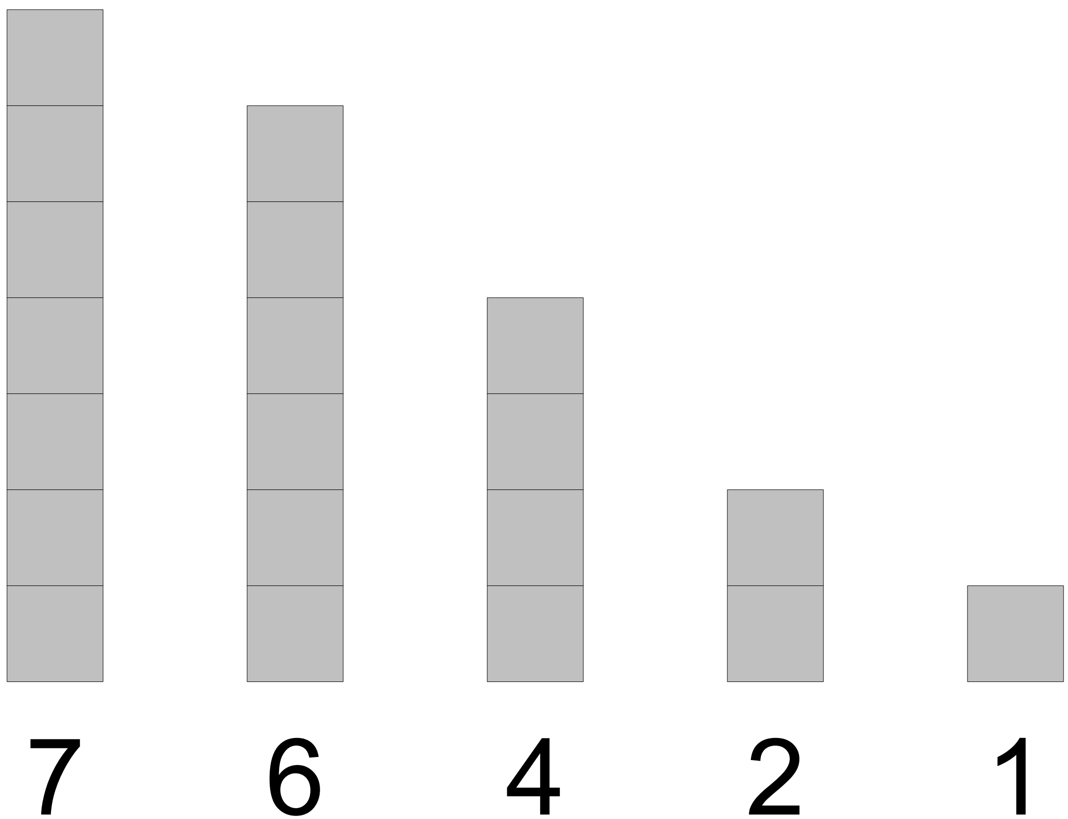
How Fast is Insertion Sort?



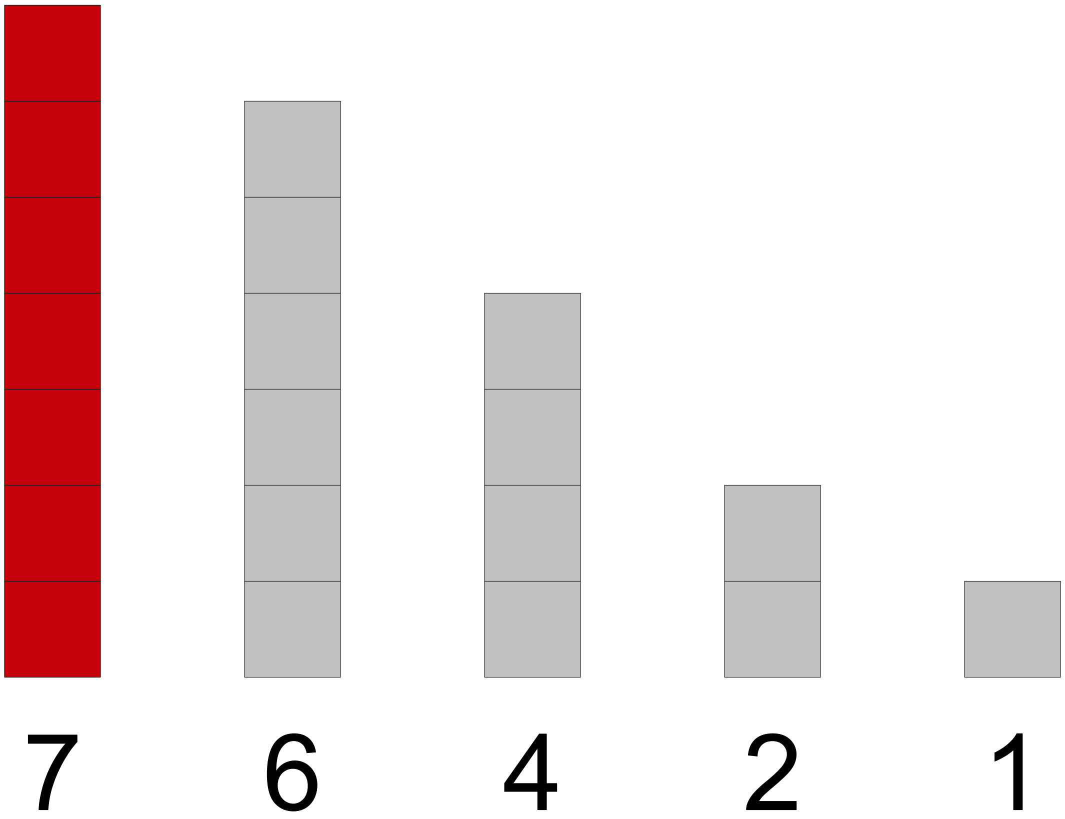
1 2 4 6 7

Work done: **O(*n*)**

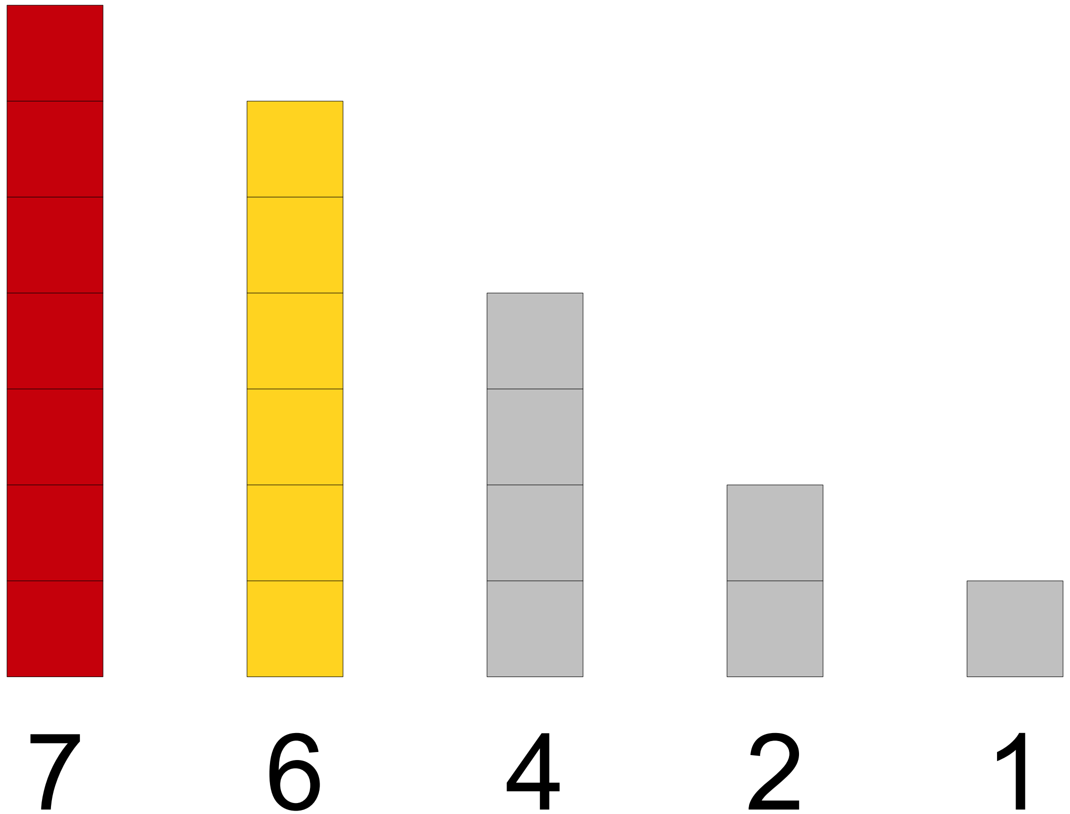
How Fast is Insertion Sort?



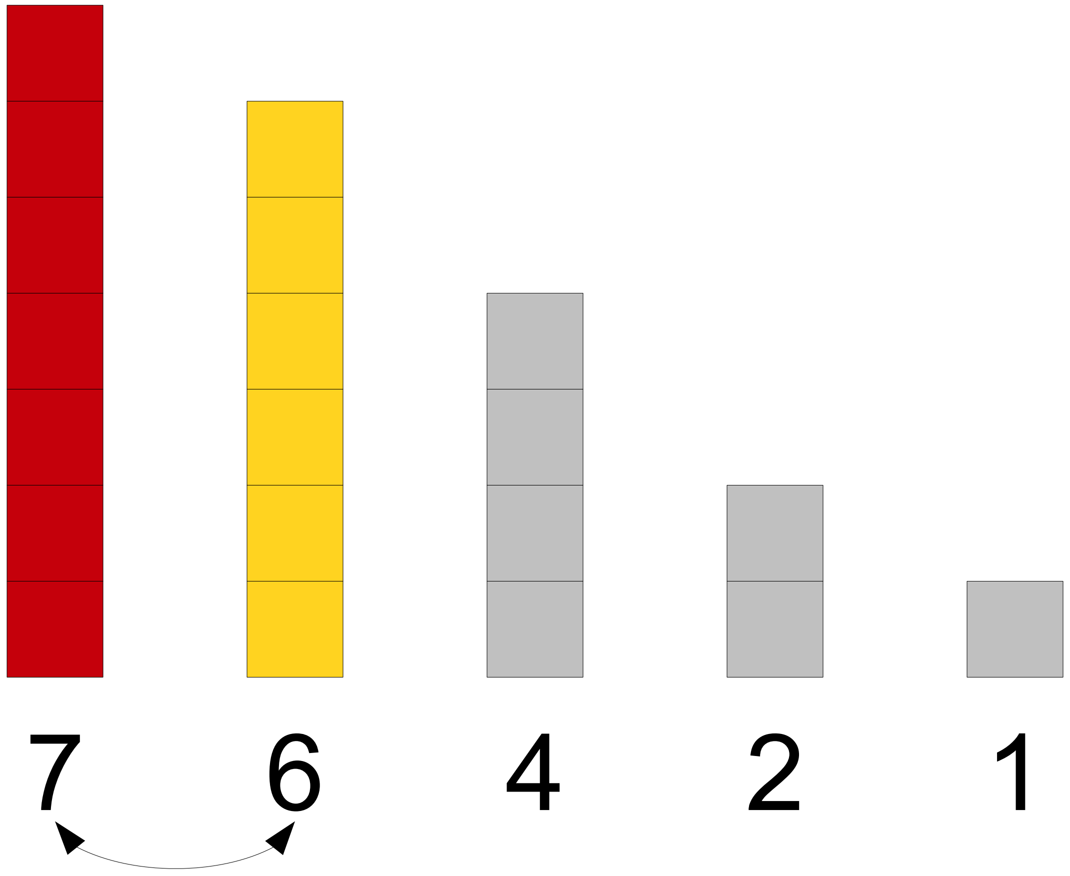
How Fast is Insertion Sort?



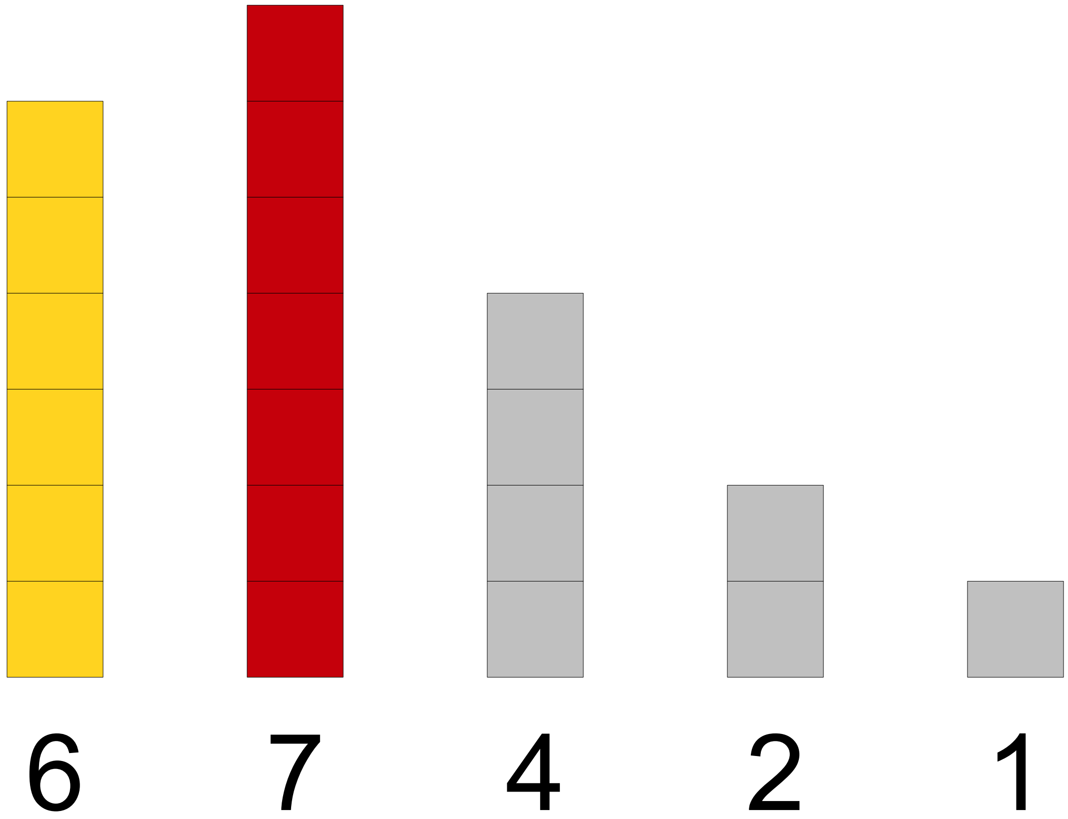
How Fast is Insertion Sort?



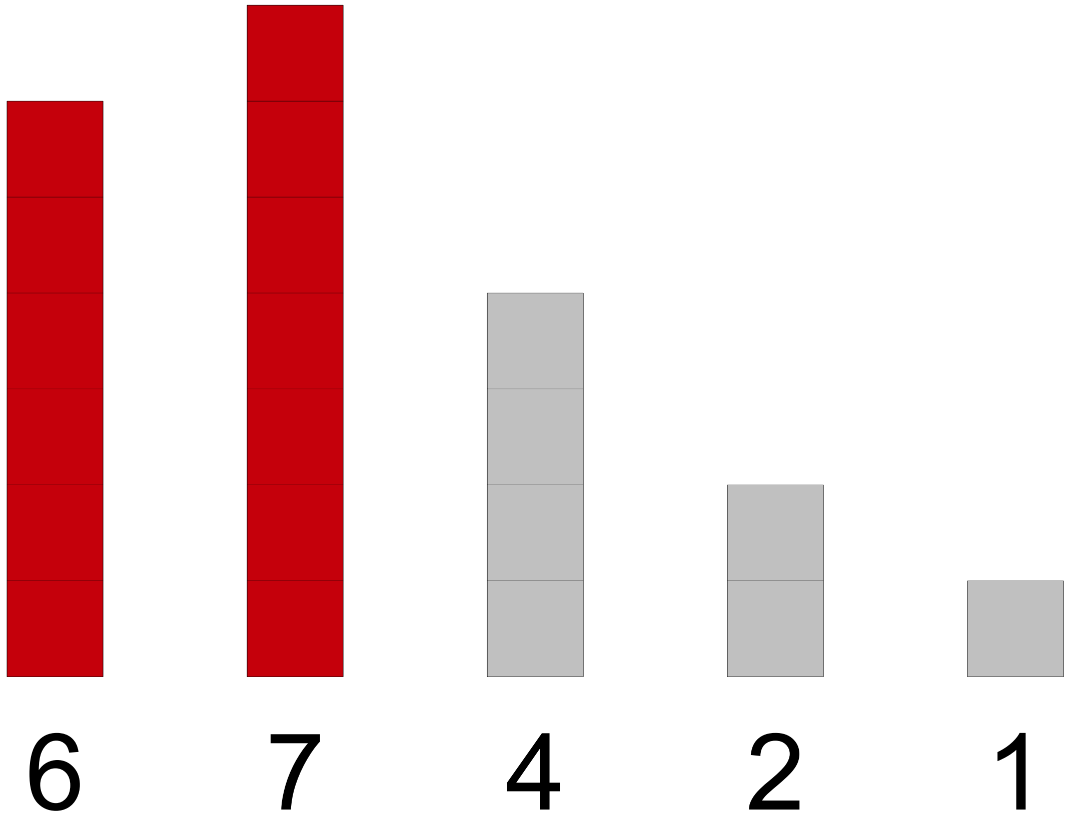
How Fast is Insertion Sort?



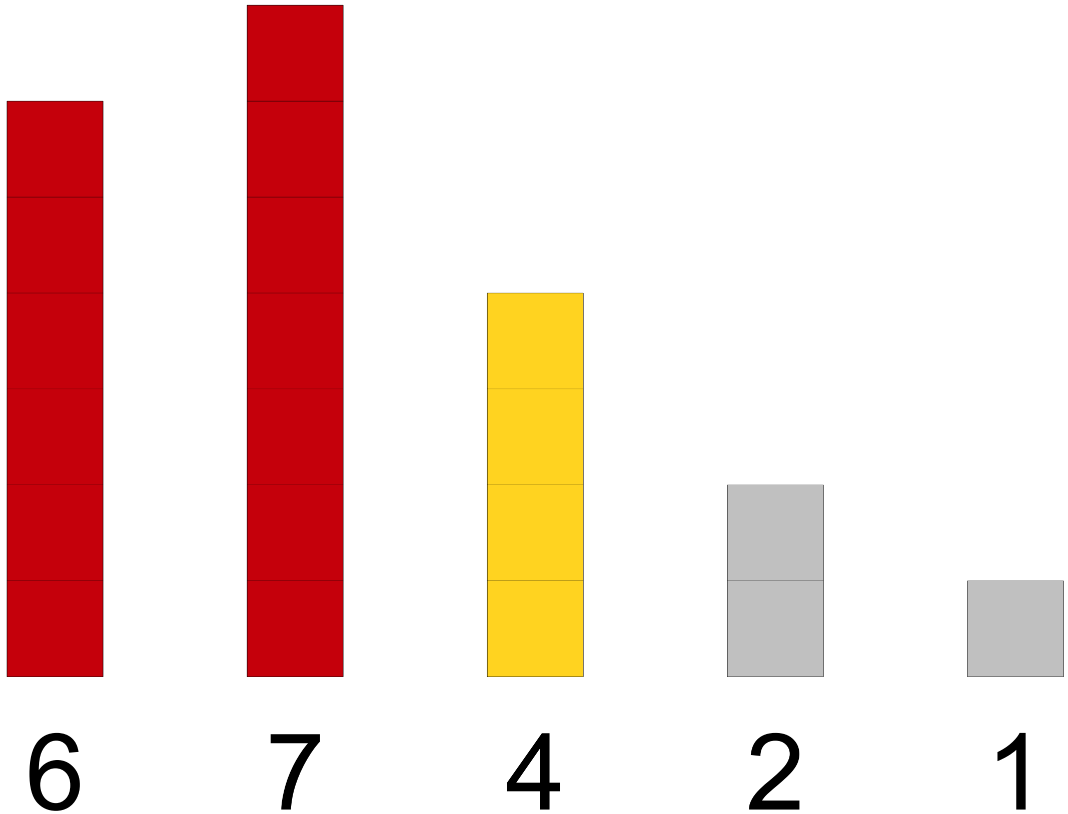
How Fast is Insertion Sort?



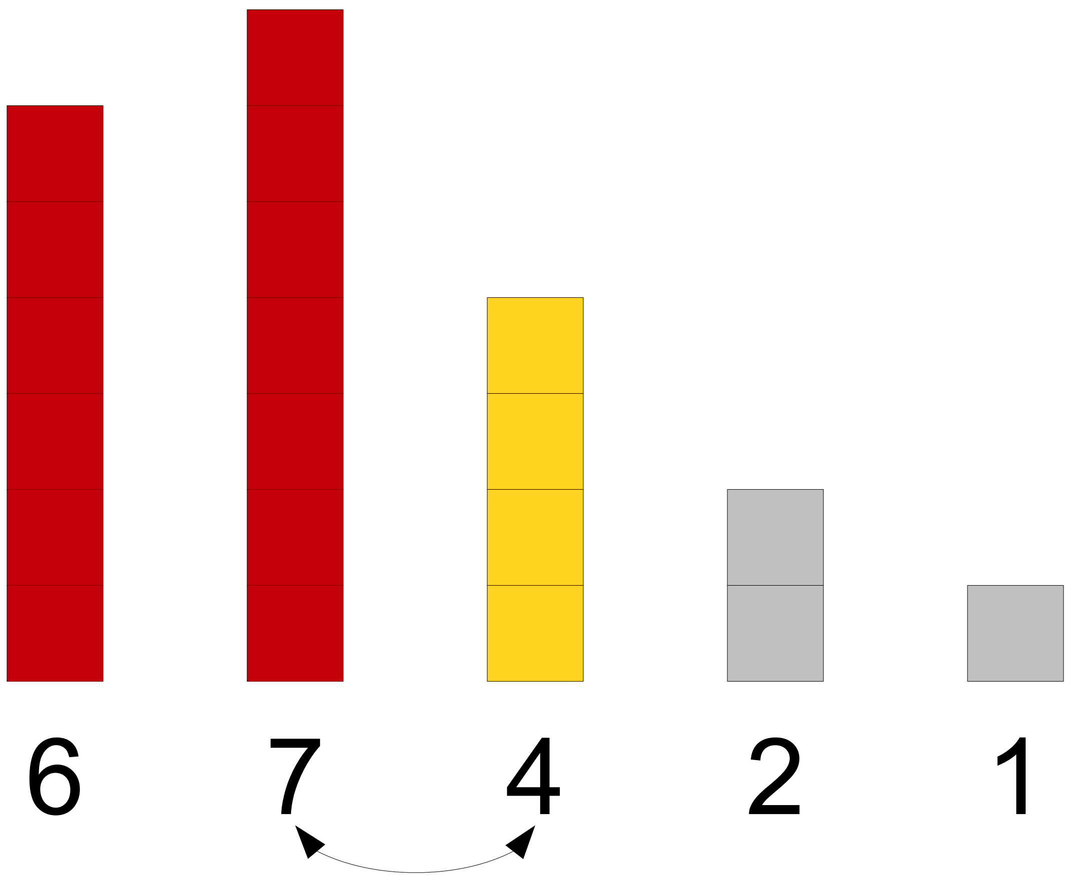
How Fast is Insertion Sort?



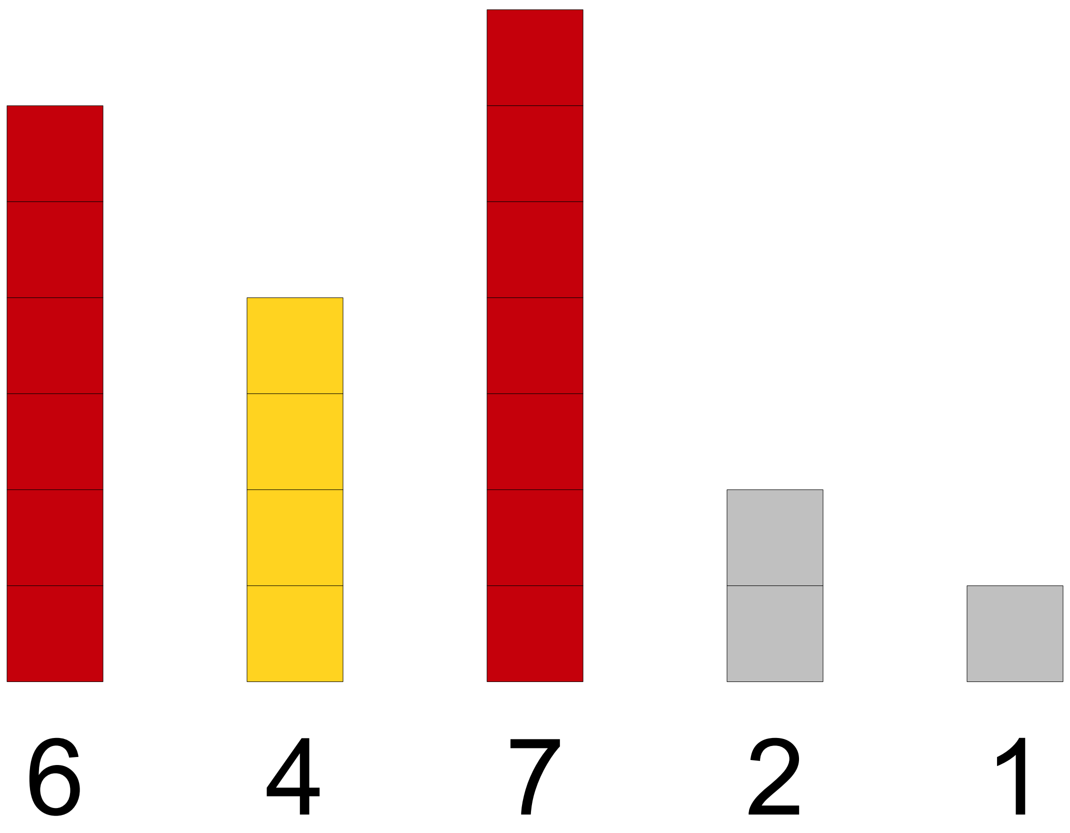
How Fast is Insertion Sort?



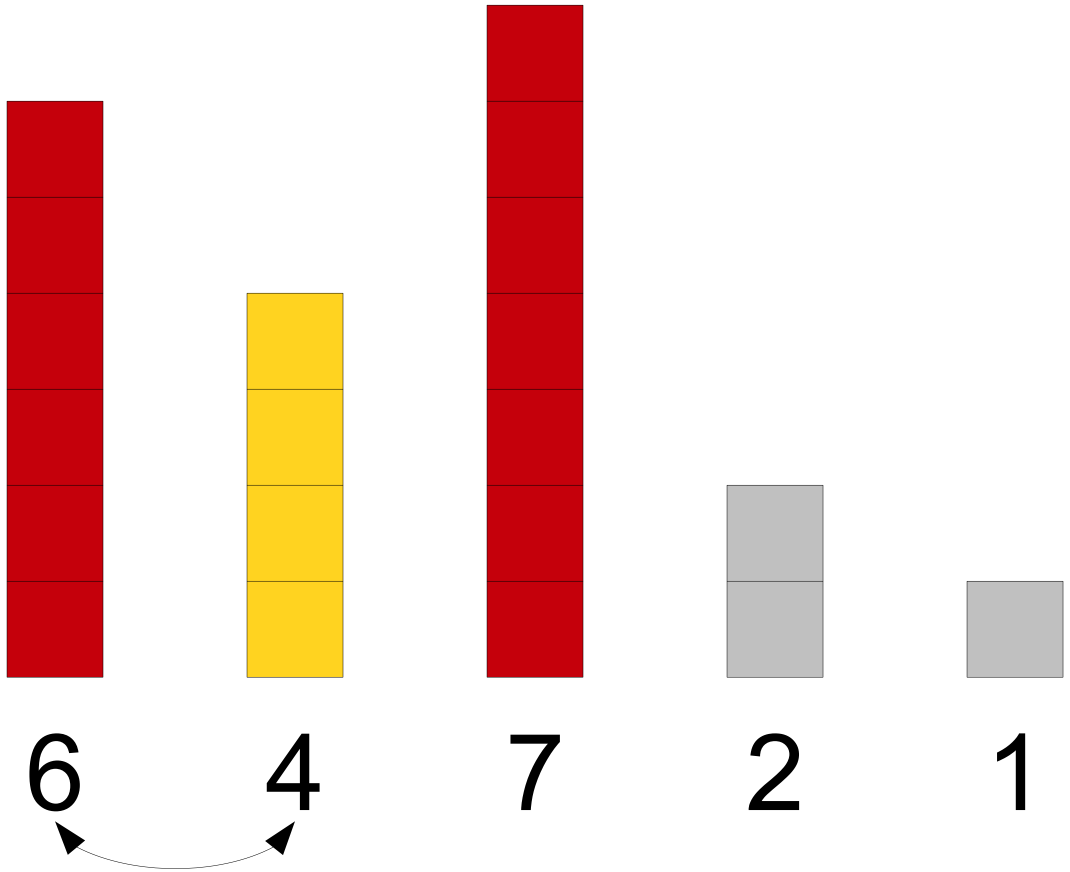
How Fast is Insertion Sort?



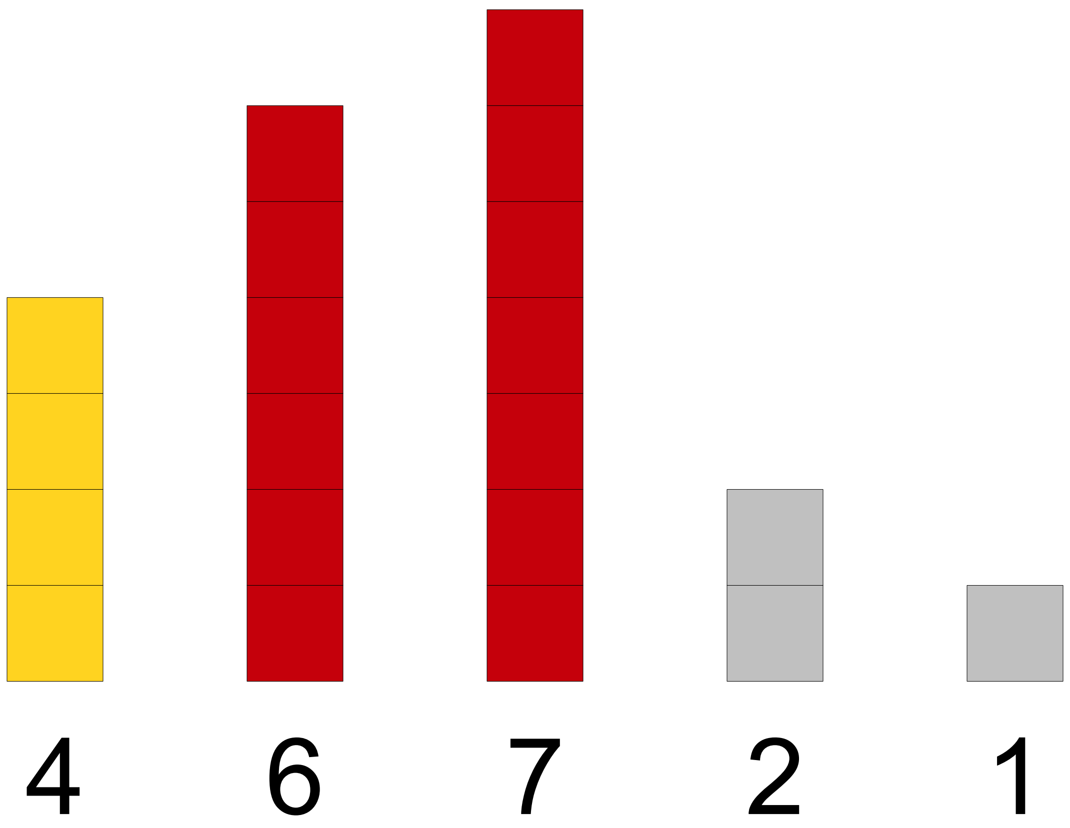
How Fast is Insertion Sort?



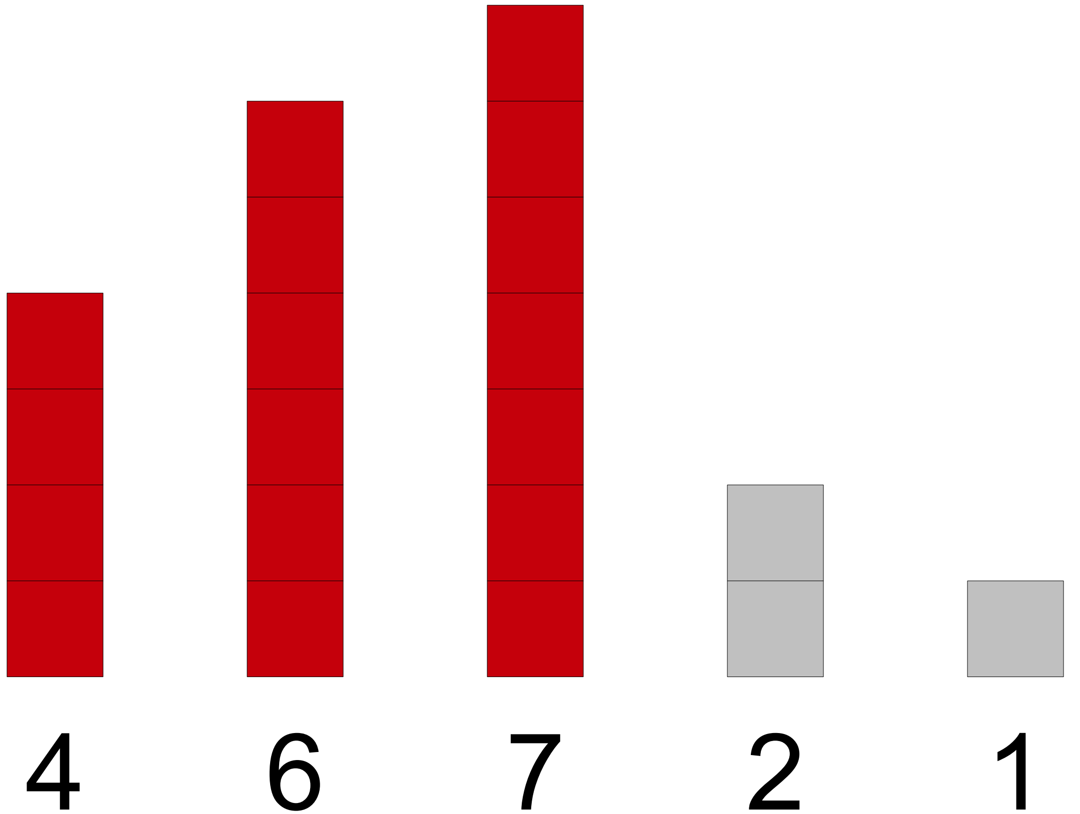
How Fast is Insertion Sort?



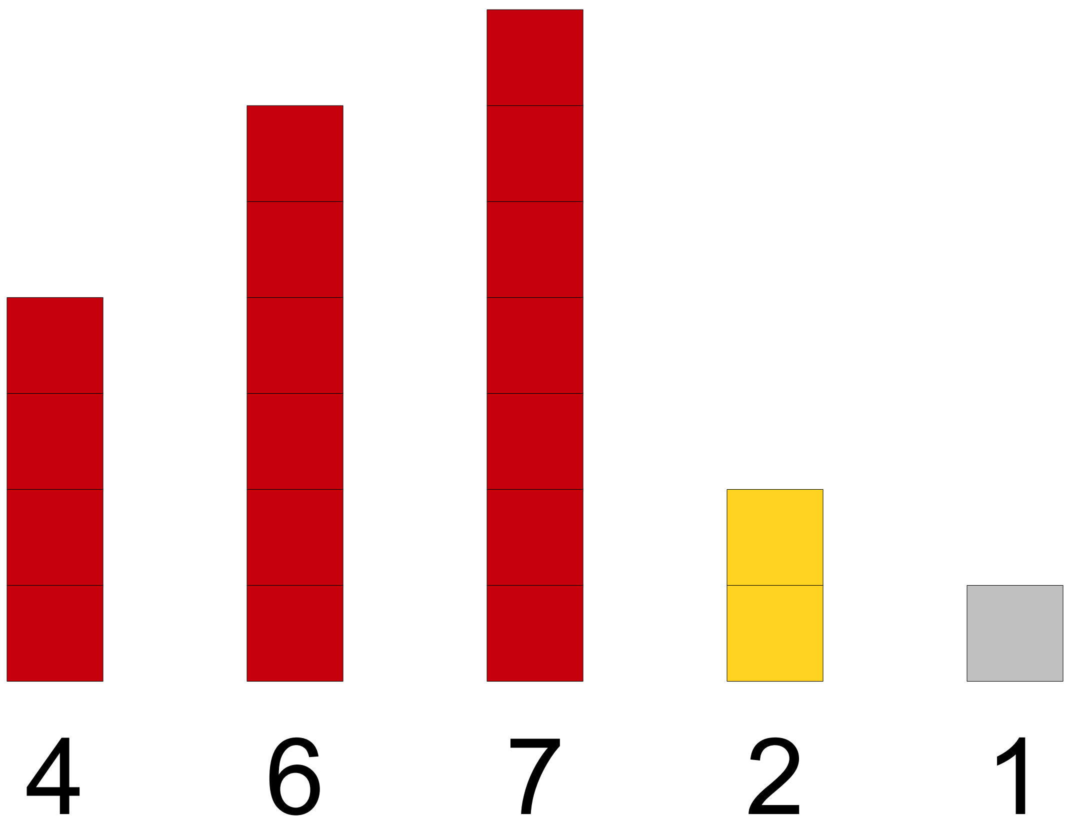
How Fast is Insertion Sort?



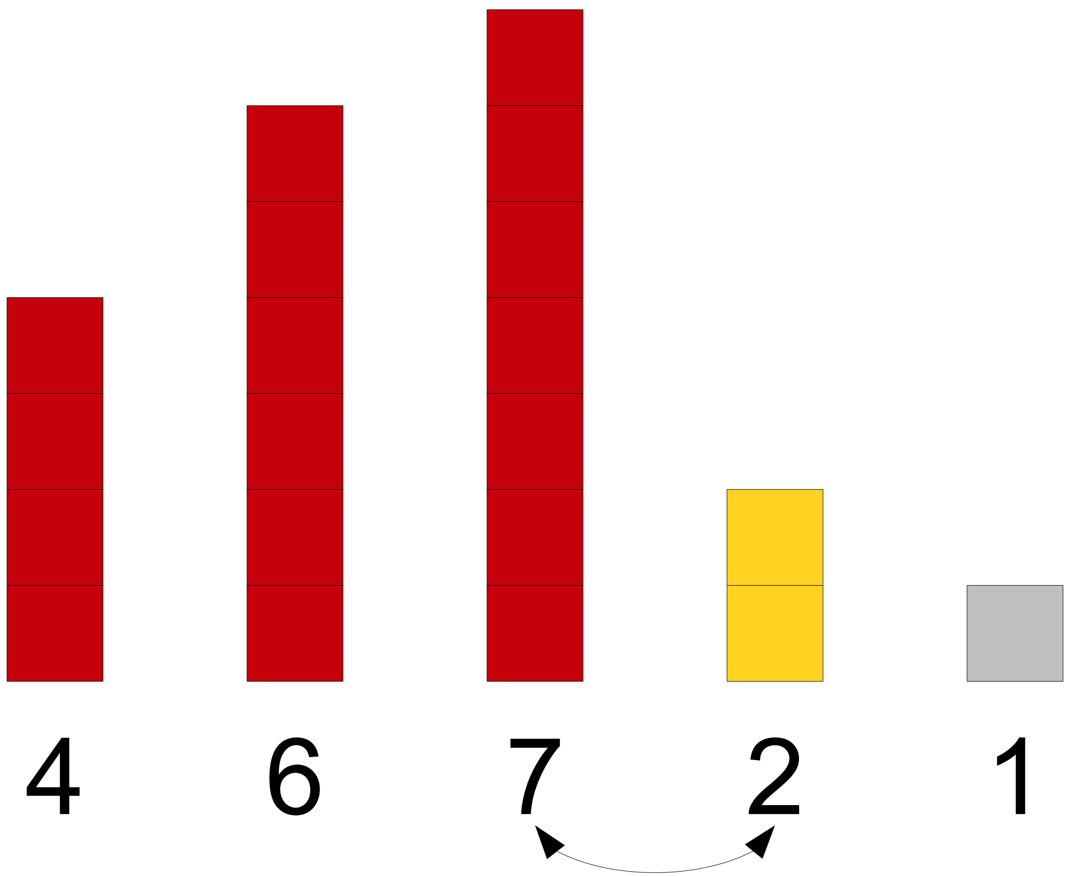
How Fast is Insertion Sort?



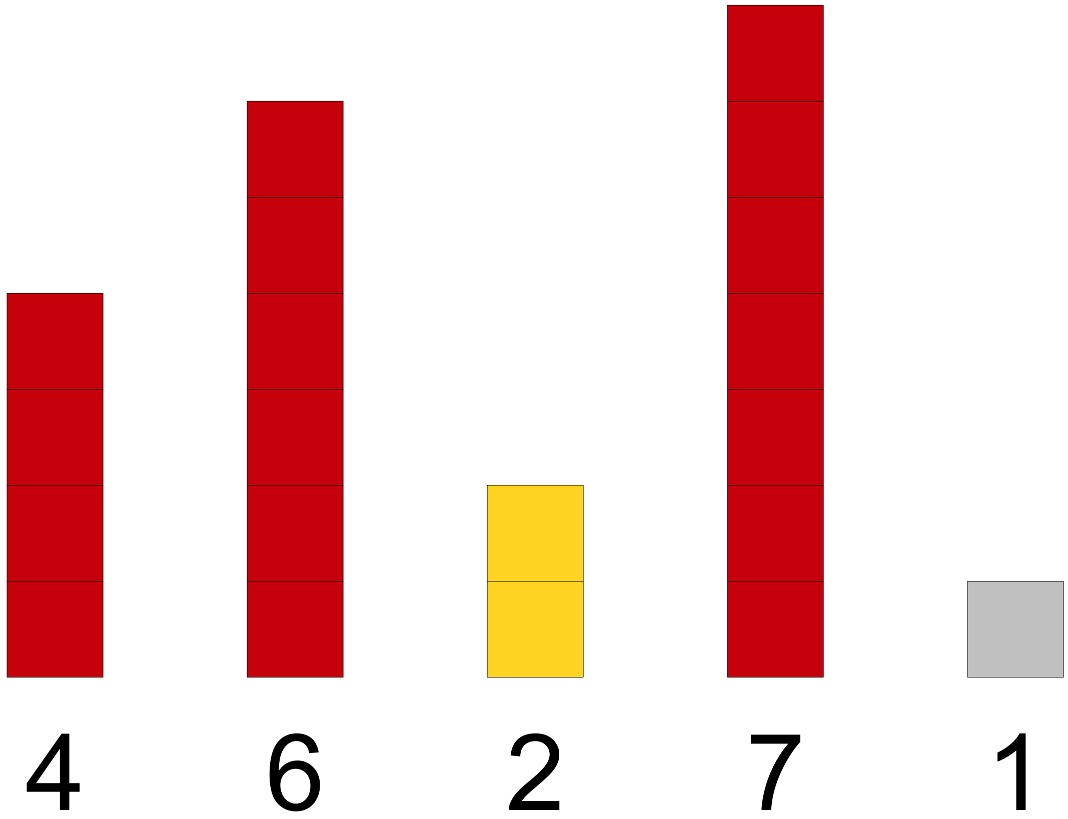
How Fast is Insertion Sort?



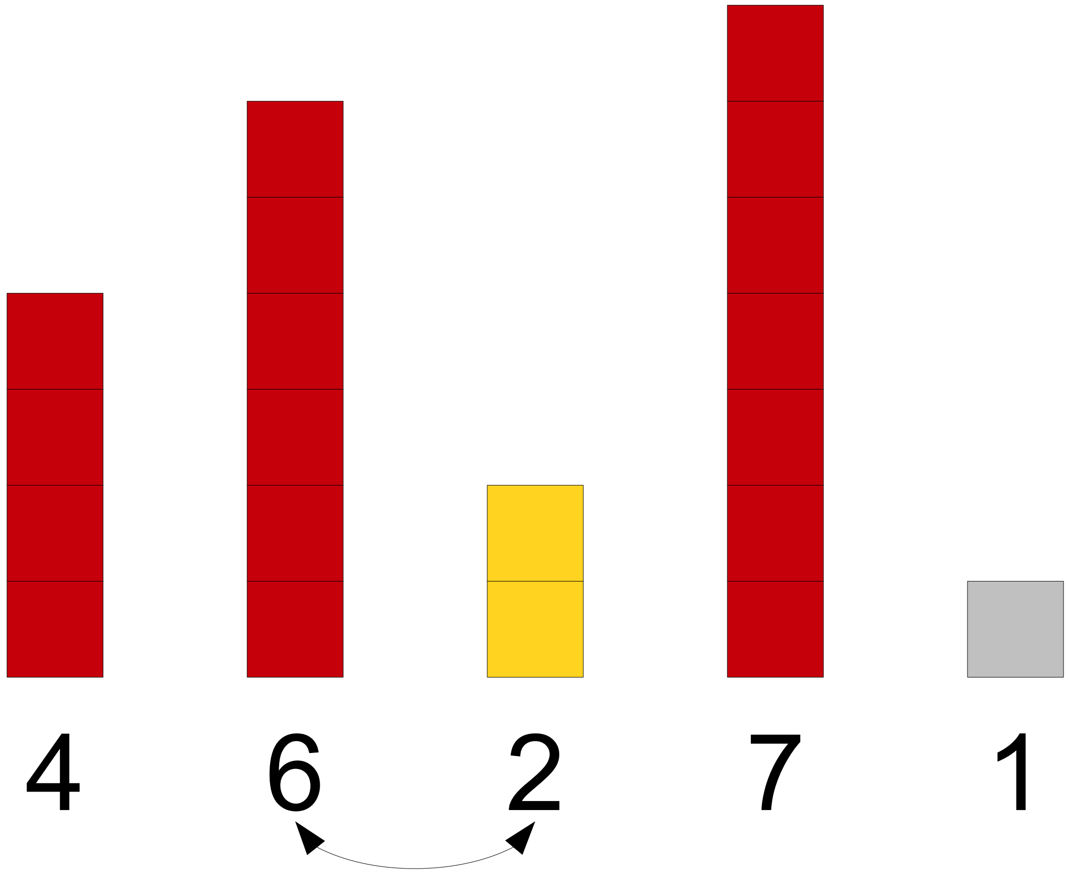
How Fast is Insertion Sort?



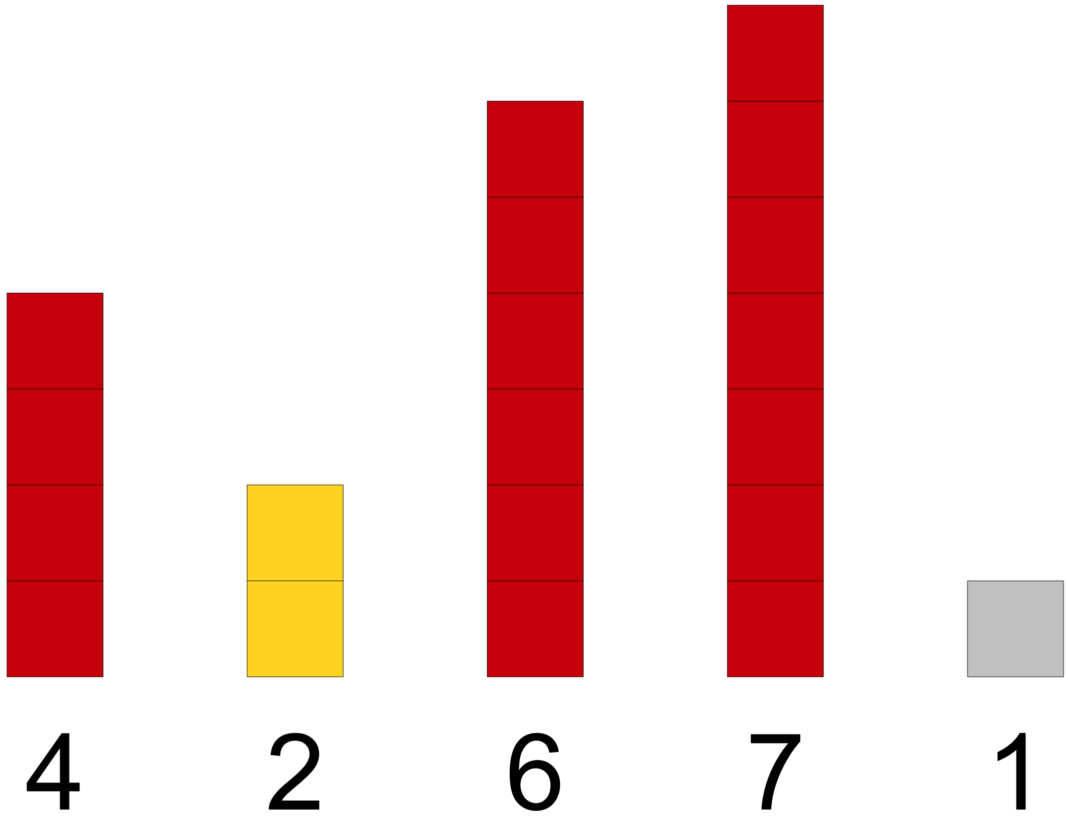
How Fast is Insertion Sort?



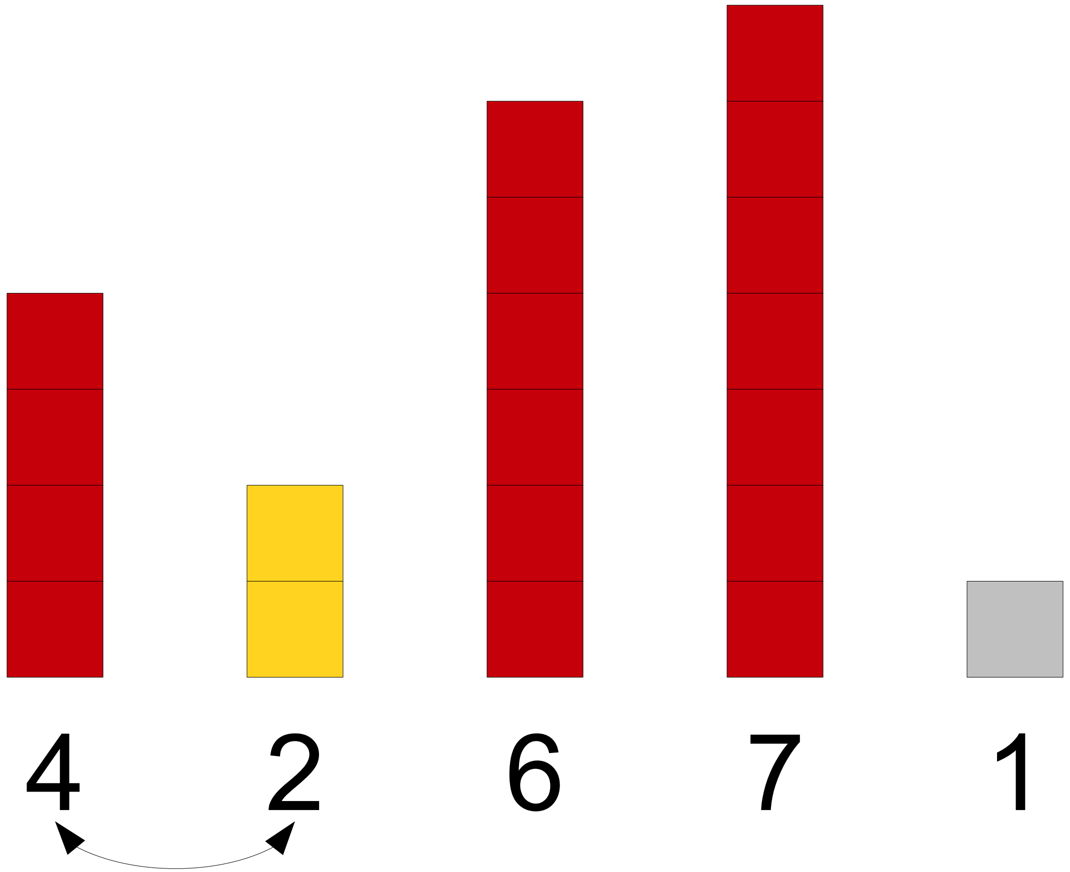
How Fast is Insertion Sort?



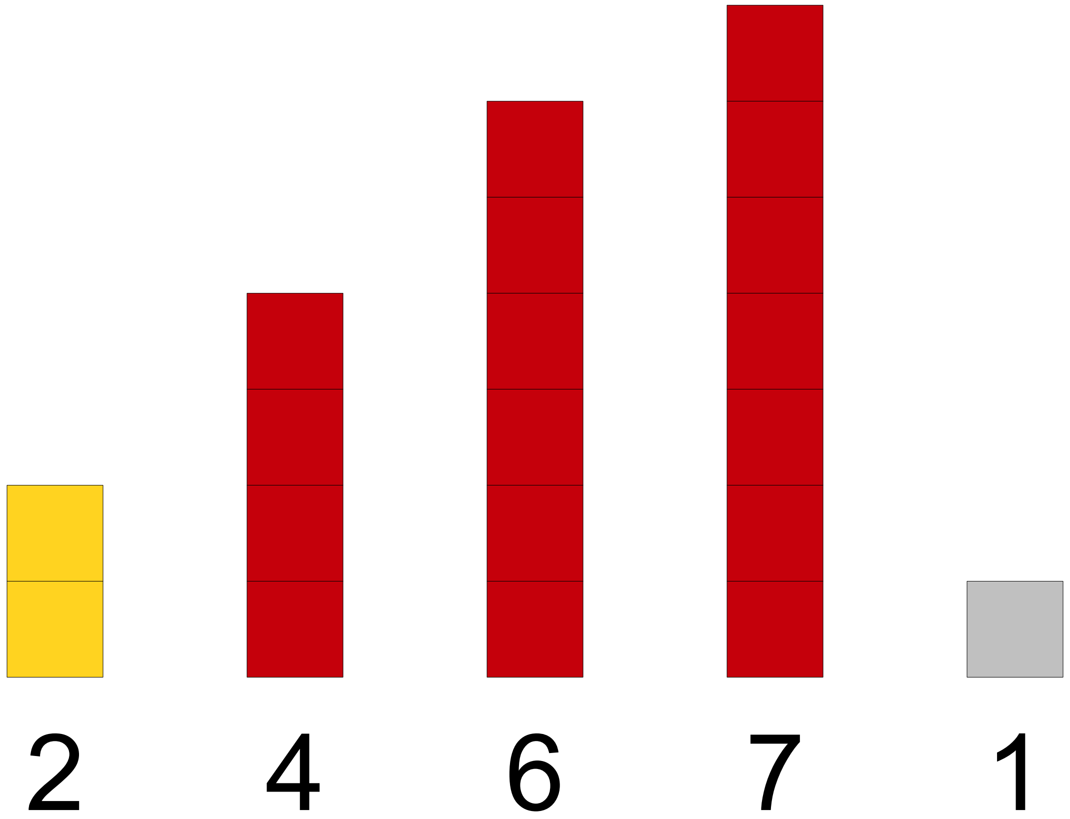
How Fast is Insertion Sort?



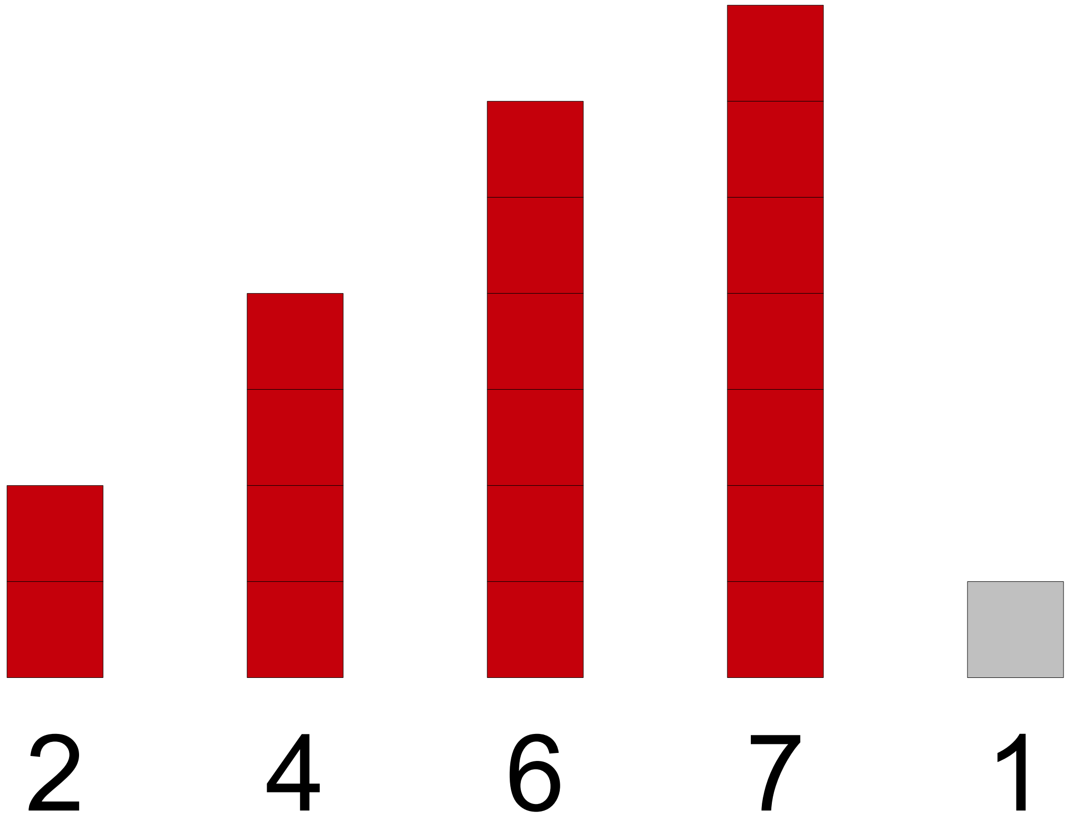
How Fast is Insertion Sort?



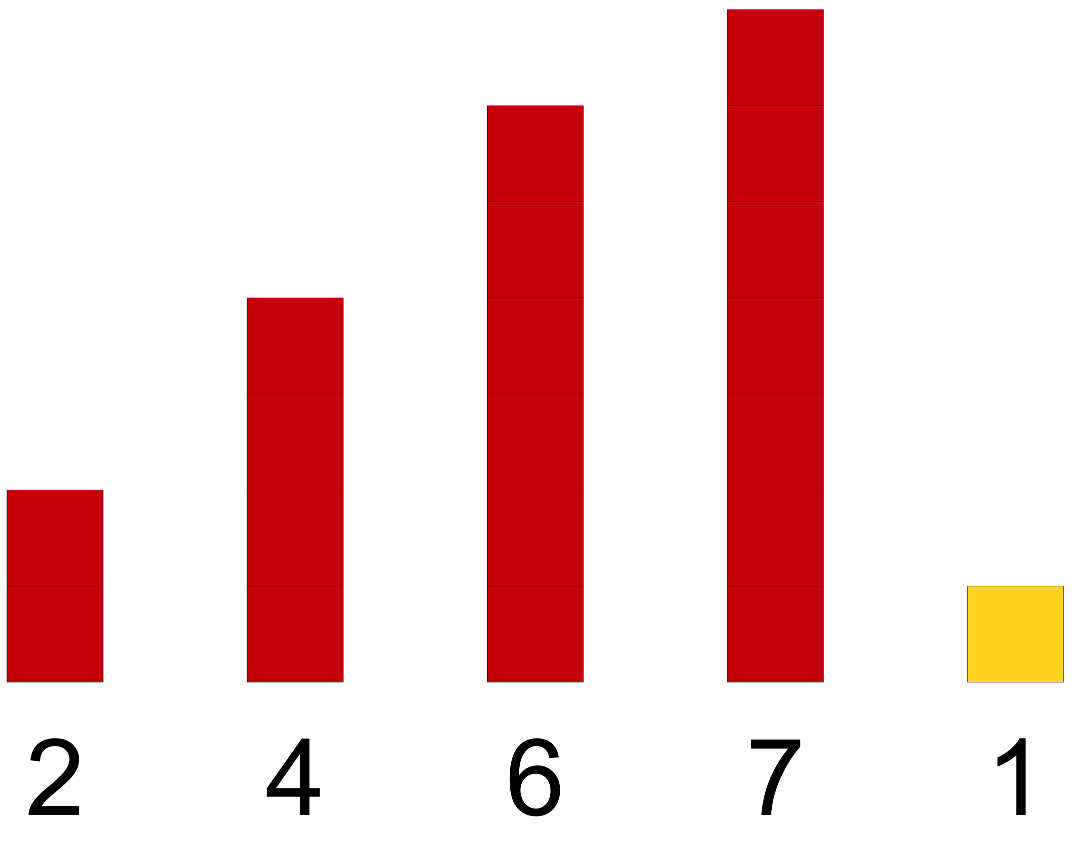
How Fast is Insertion Sort?



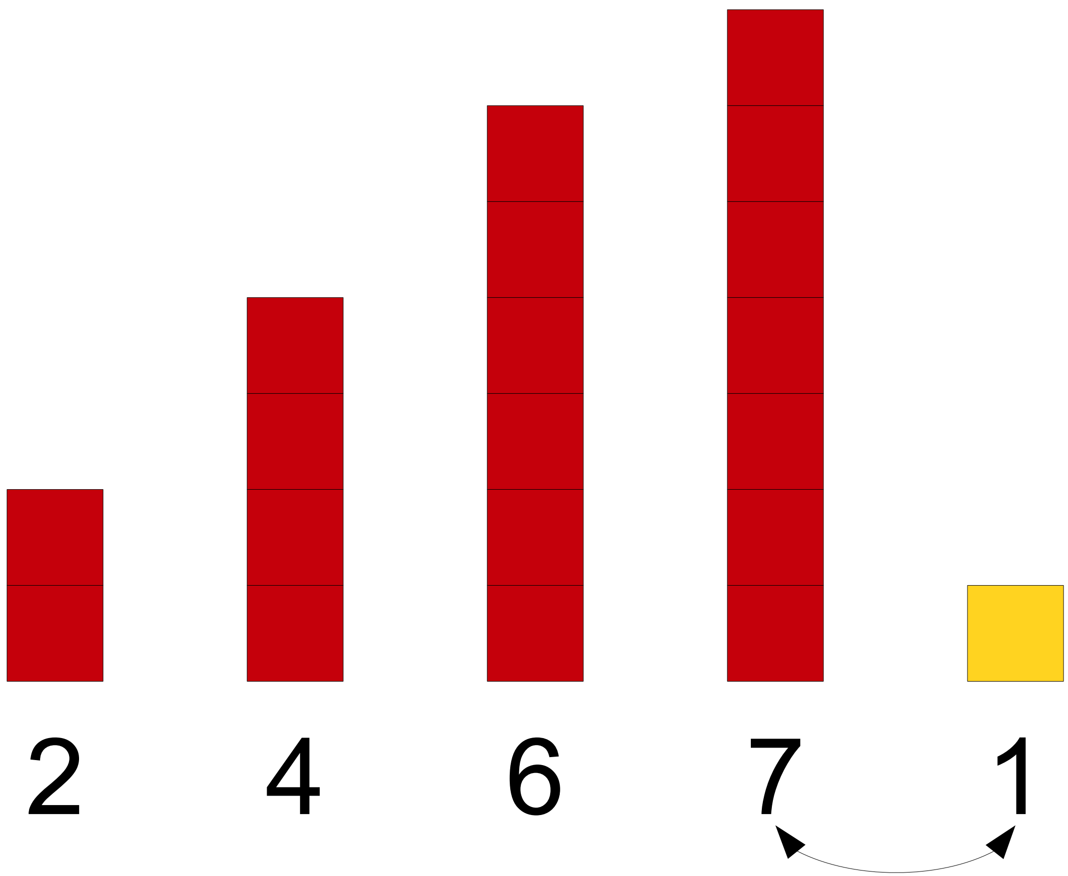
How Fast is Insertion Sort?



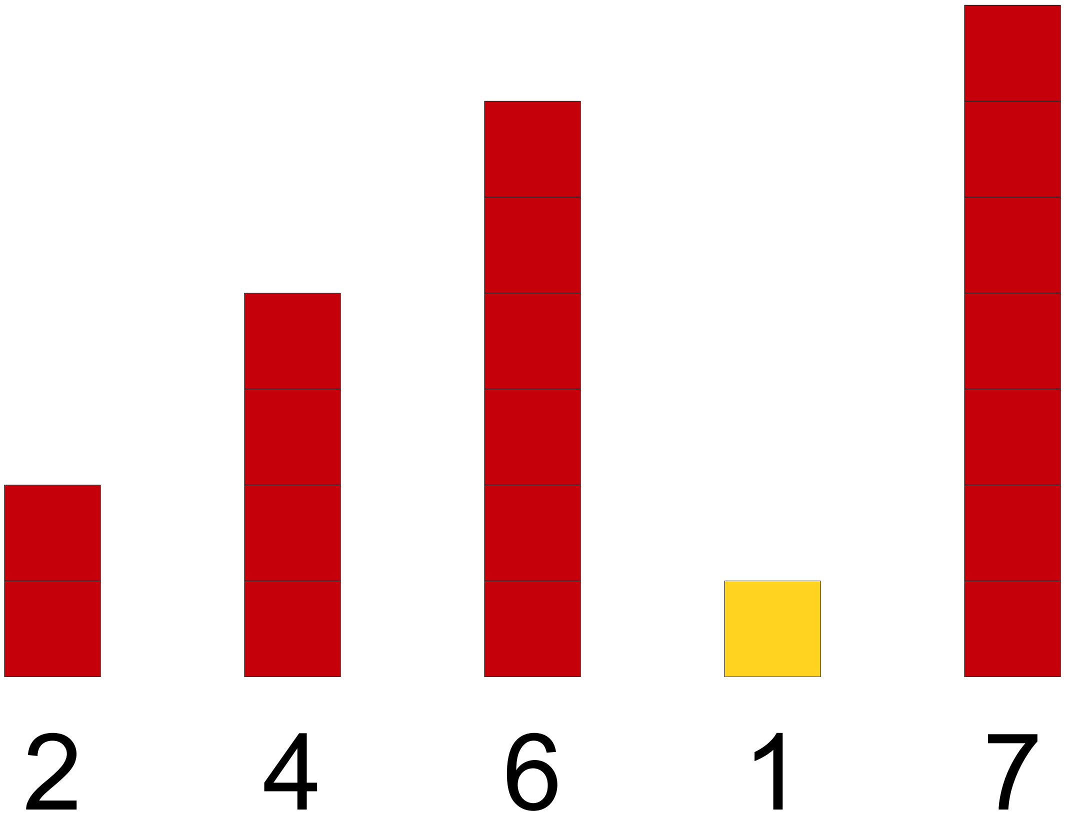
How Fast is Insertion Sort?



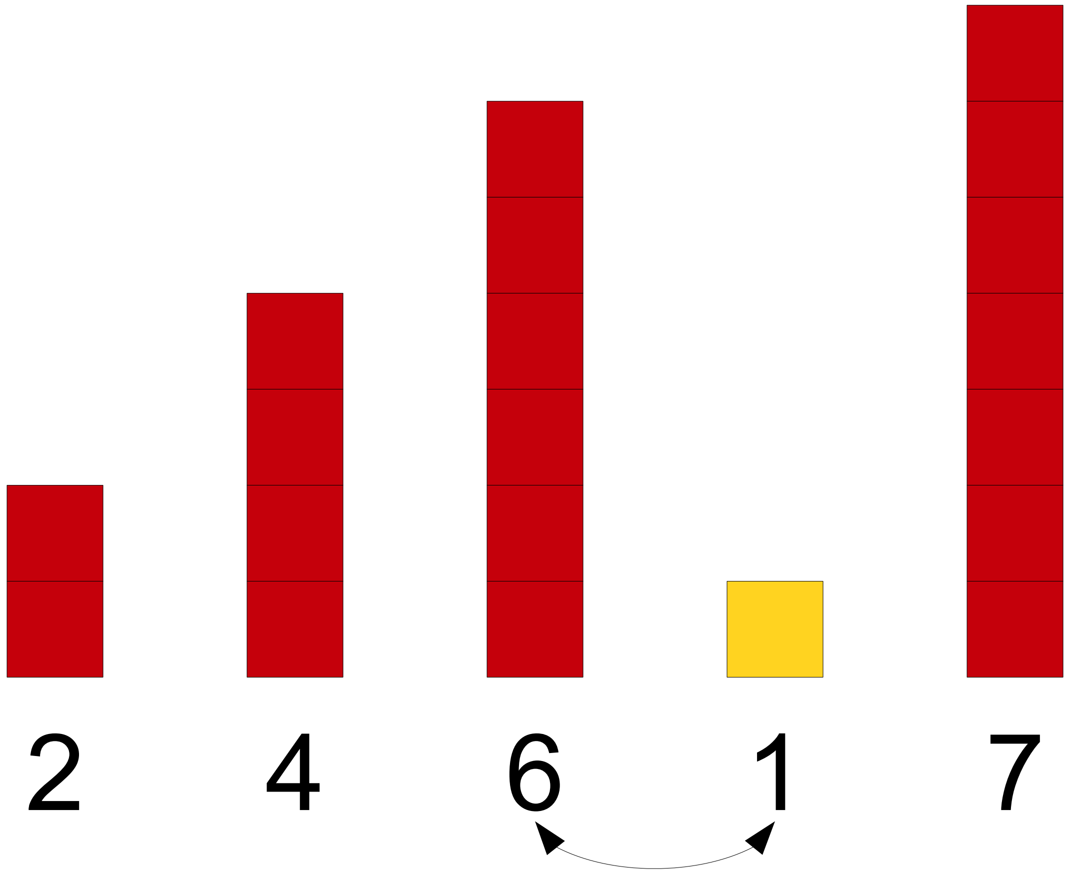
How Fast is Insertion Sort?



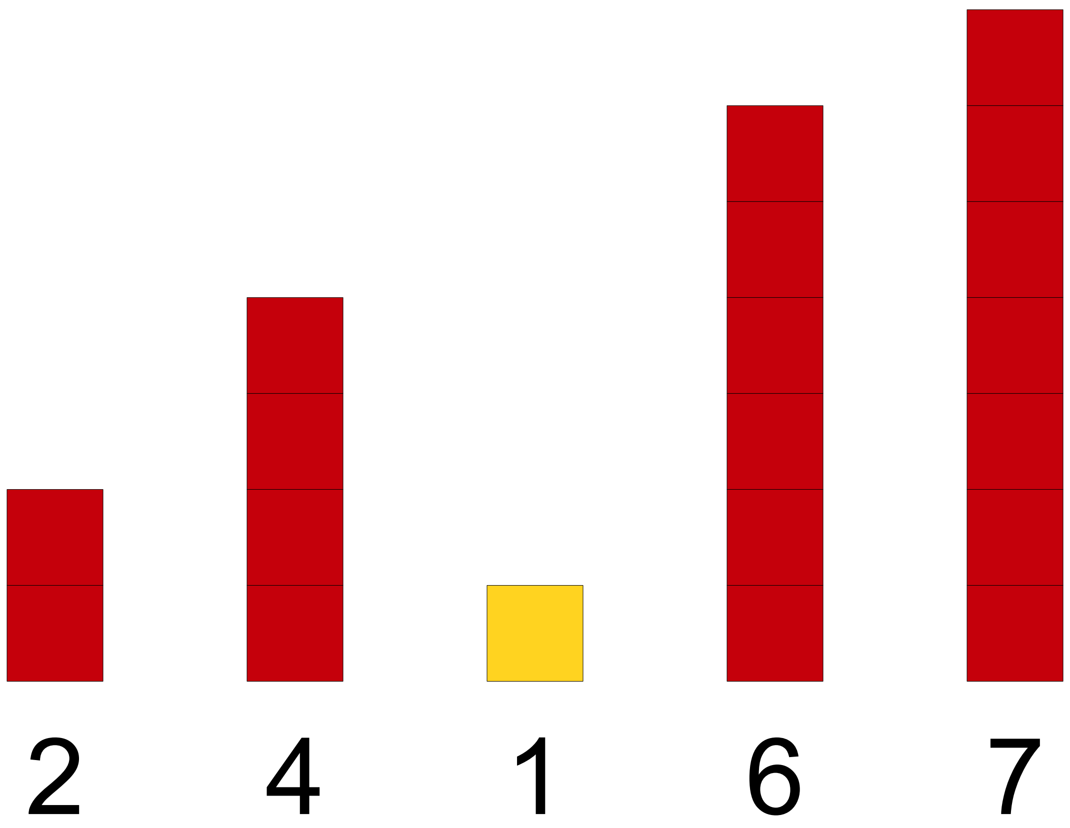
How Fast is Insertion Sort?



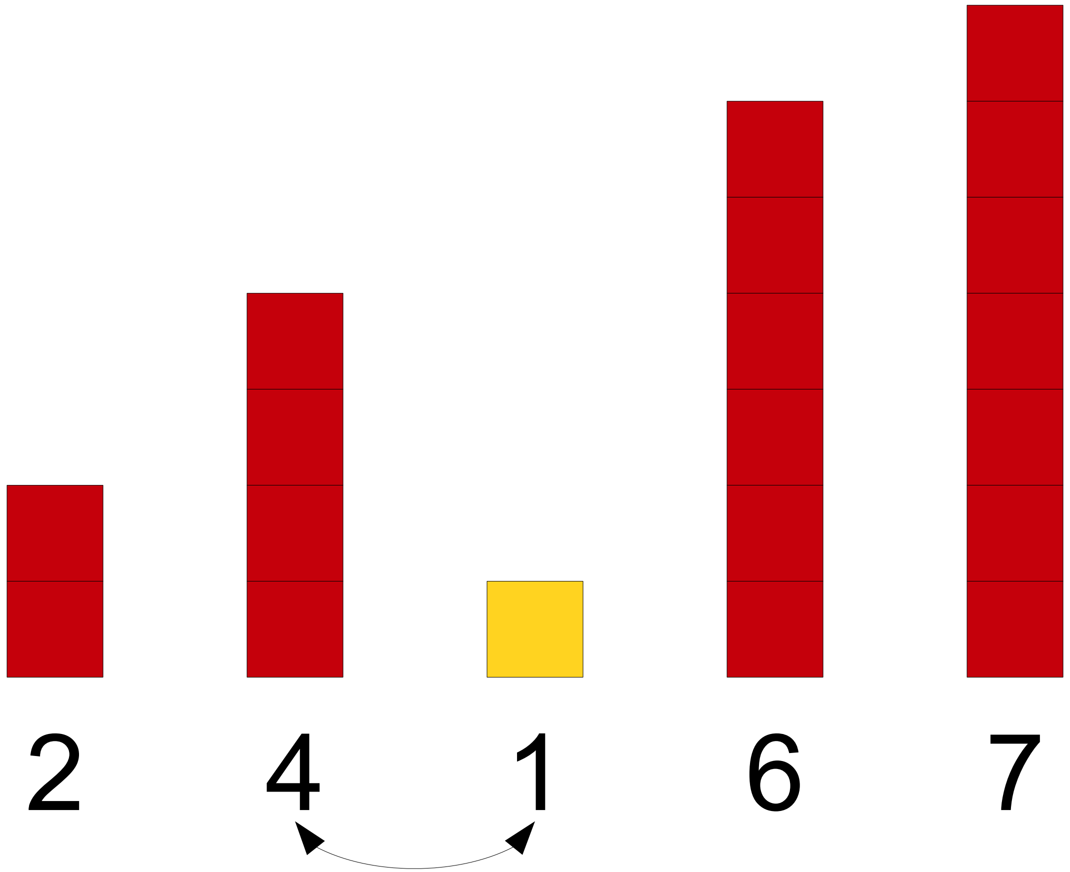
How Fast is Insertion Sort?



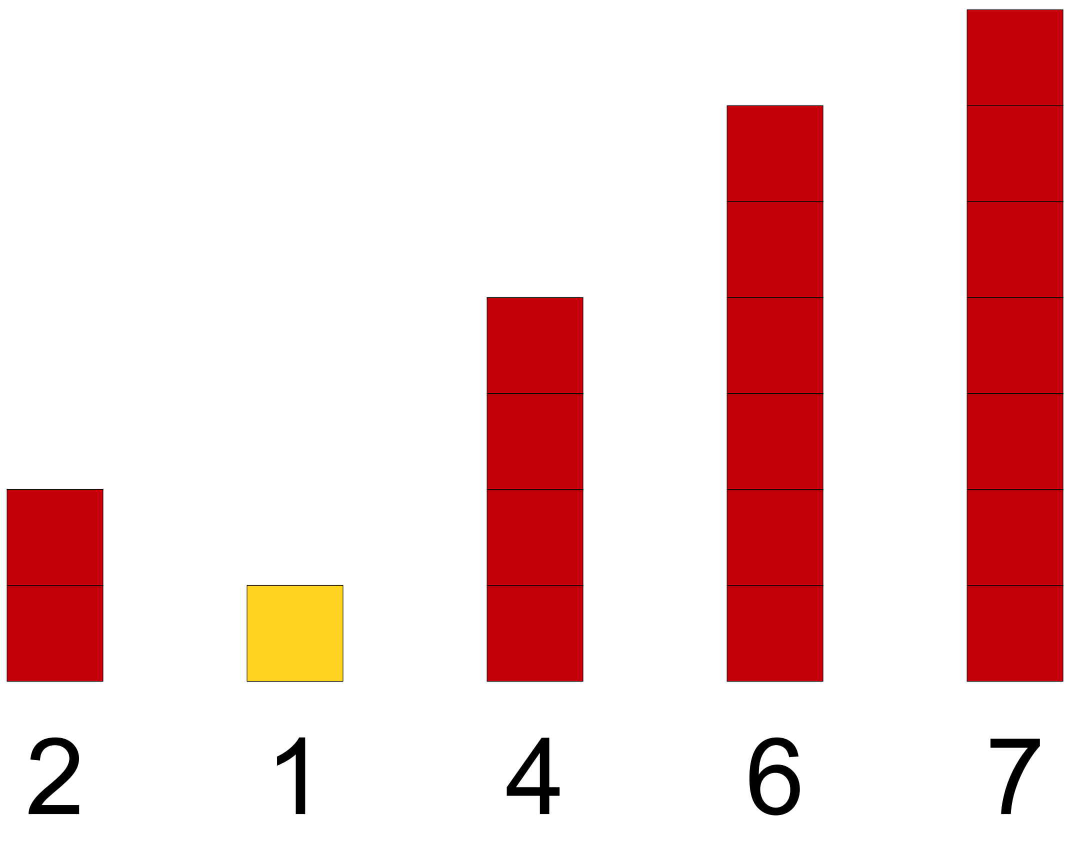
How Fast is Insertion Sort?



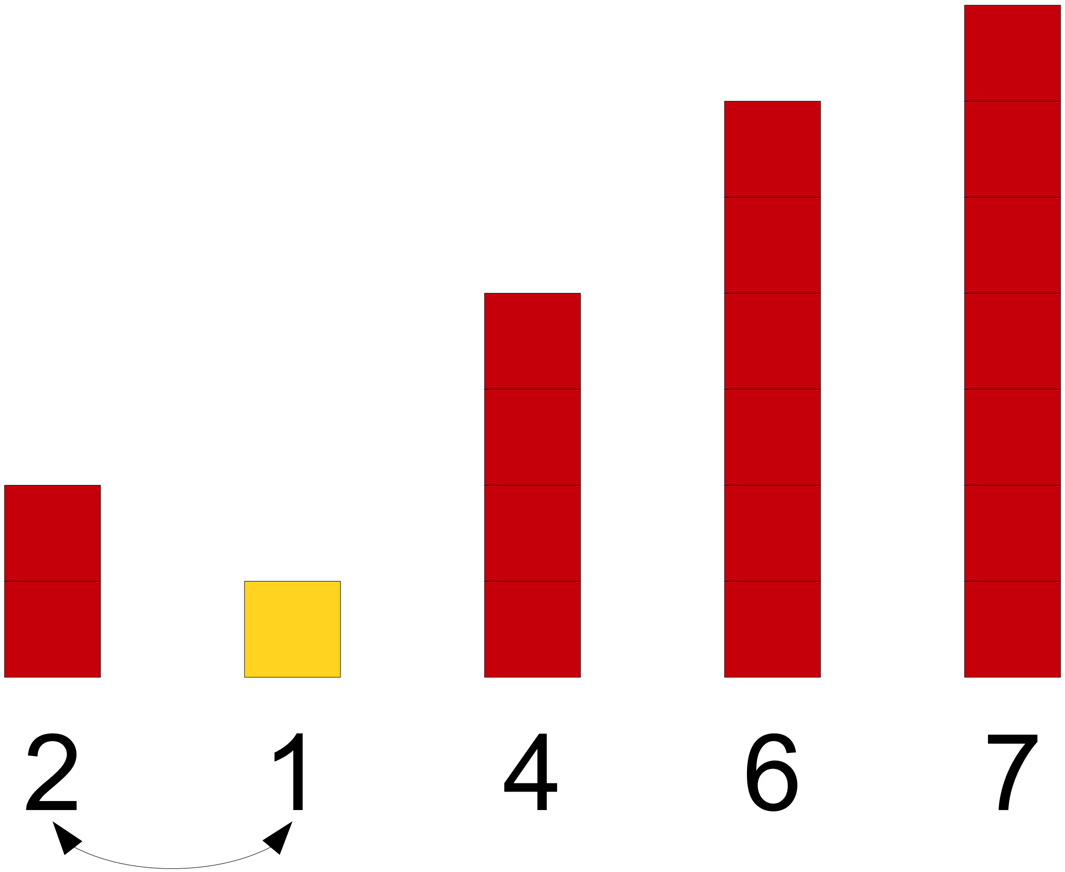
How Fast is Insertion Sort?



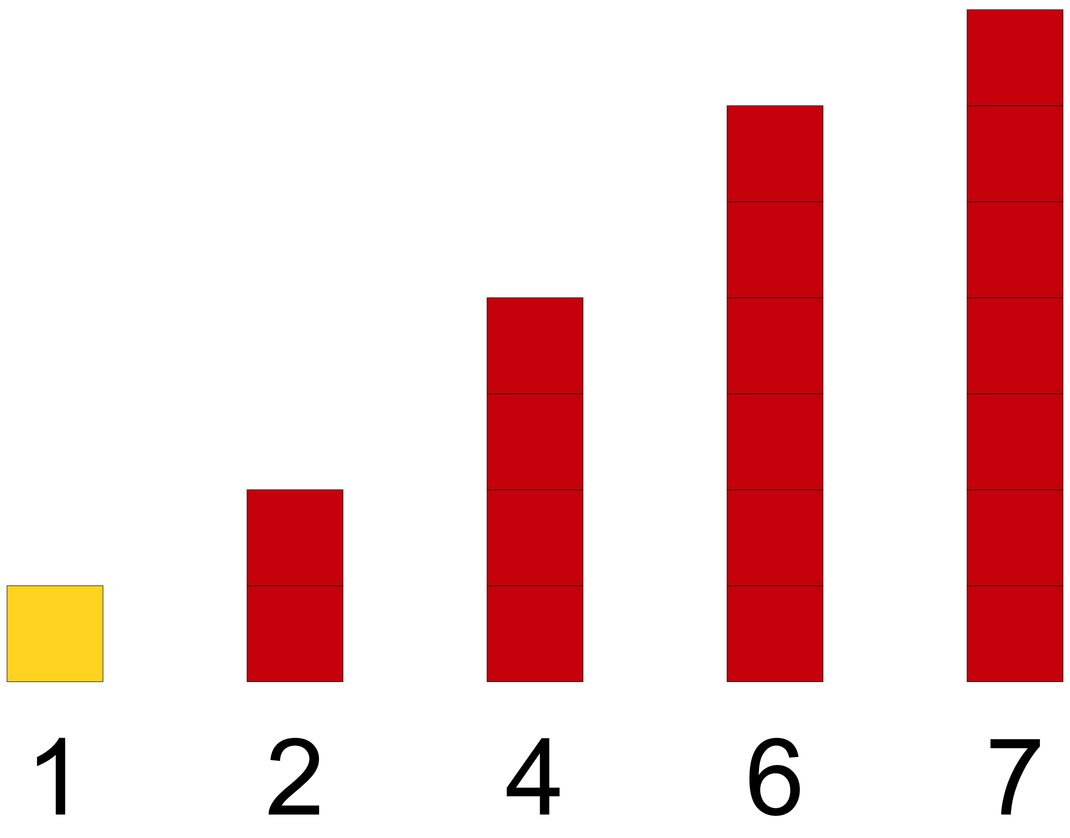
How Fast is Insertion Sort?



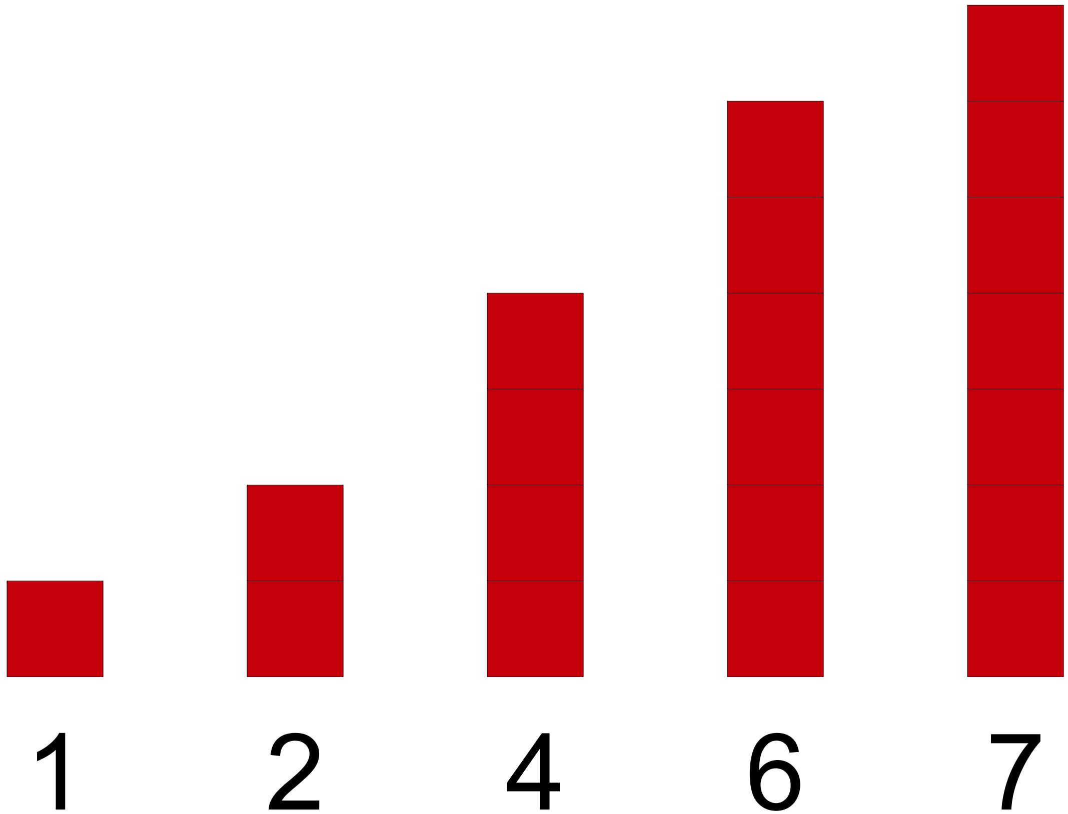
How Fast is Insertion Sort?



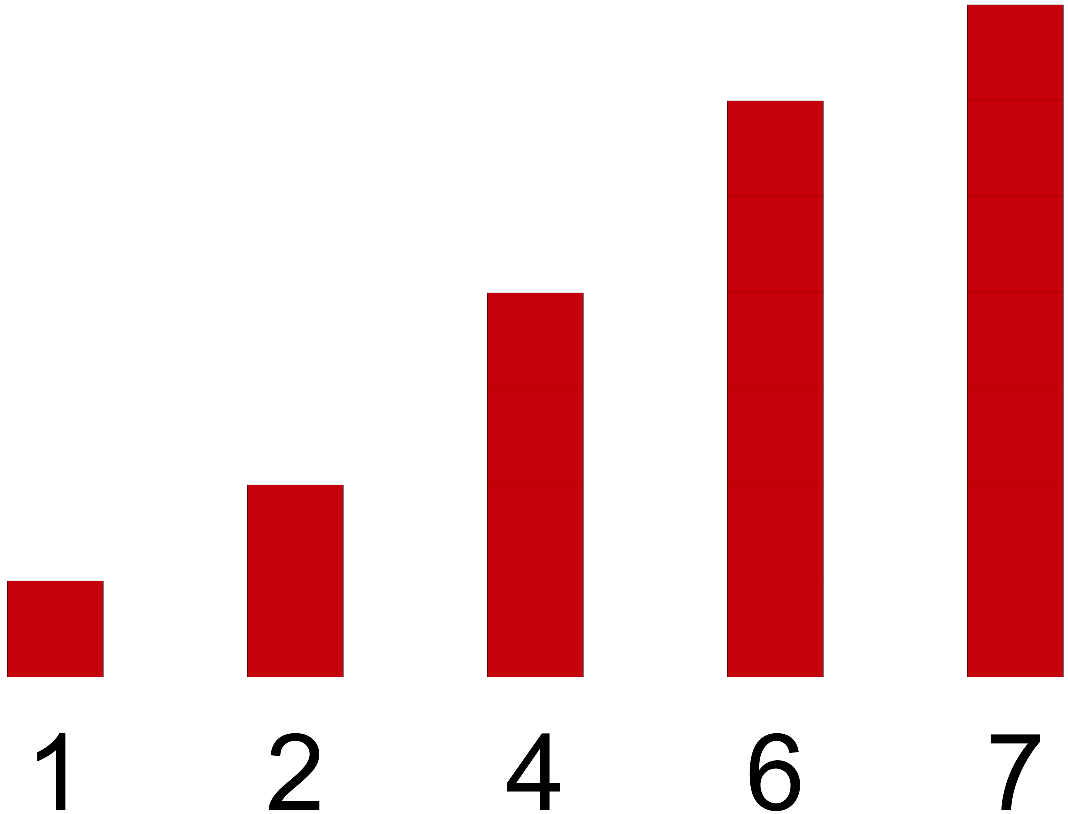
How Fast is Insertion Sort?



How Fast is Insertion Sort?



How Fast is Insertion Sort?



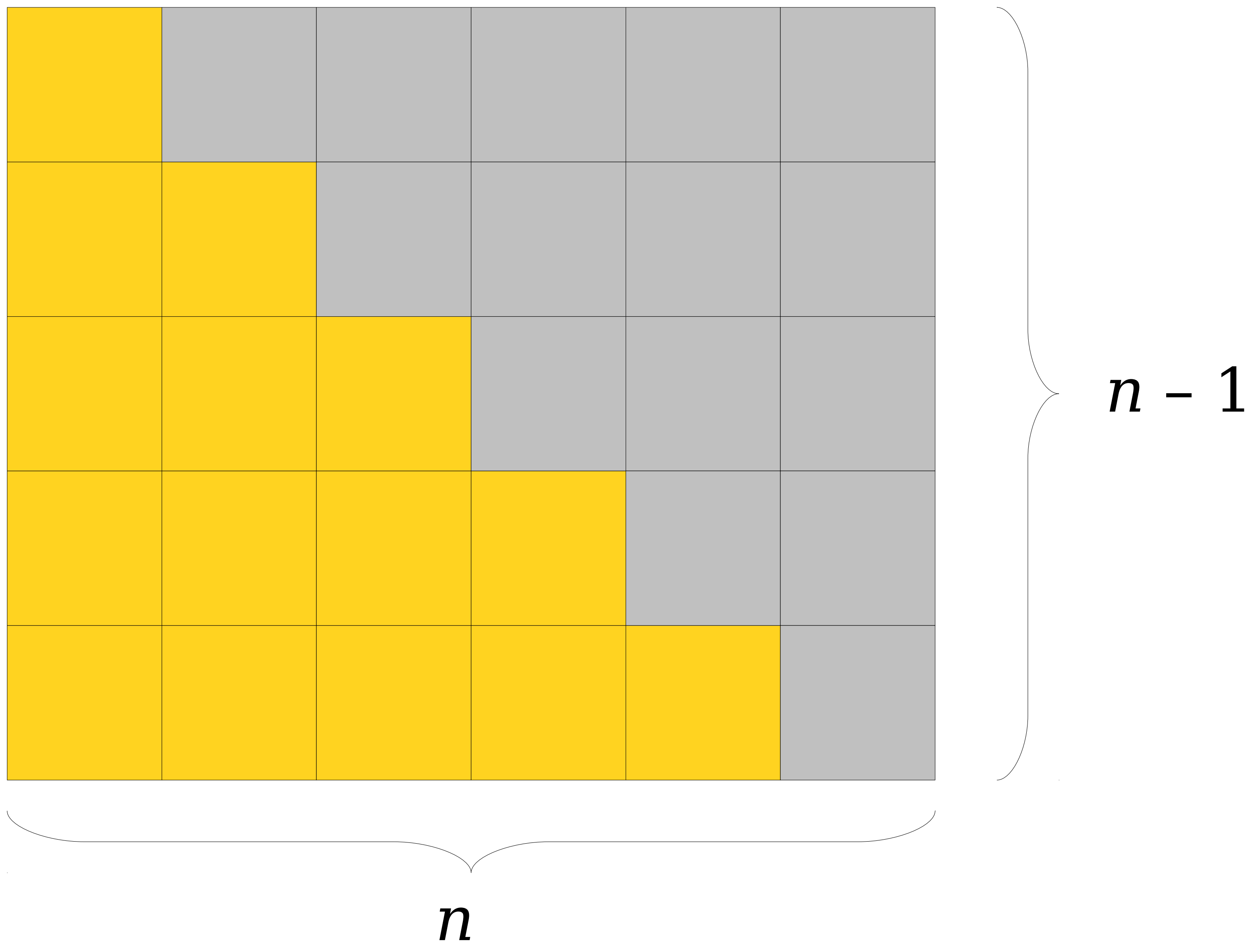
Work Done: **1 + 2 + 3 + 4**

If we run insertion sort on a sequence of n elements, we might have to do

$$1 + 2 + 3 + 4 + \ldots + (n - 2) + (n - 1)$$

swaps. How many swaps is this?

$$1 + 2 + 3 + \ldots + (n - 2) + (n - 1) = n(n - 1) / 2$$



The Complexity of Insertion Sort

- In the worst case, insertion sort takes time

$$
\begin{aligned}
& O(n\ (n - 1) / 2) \\
= {} & O(n\ (n - 1)) \\
= {} & O(n^2 - n) \\
= {} & \mathbf{O(n^2)}.
\end{aligned}
$$

- ***Fun fact:*** Insertion sorting an array of random values takes, on average, $O(n^2)$ time.
  - Curious why? Come talk to me after class!

Thinking About O(n^2)

14	6	3	9	7	16	2	15

$\mathrm{T}(n)$

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

$\mathrm{T}(2n) \approx 4\mathrm{T}(n)$

Next Time

- ***Faster Sorting Algorithms***
  - Can you beat $O(n^2)$ time?
- ***Hybrid Sorting Algorithms***
  - When might insertion sort be useful?