Welcome to CS161!

- Two Handouts
- Today:
  - Course Overview
  - Correctness Proofs
  - Runtime Analysis

Course Staff

Keith Schwarz (htiek@cs.stanford.edu)

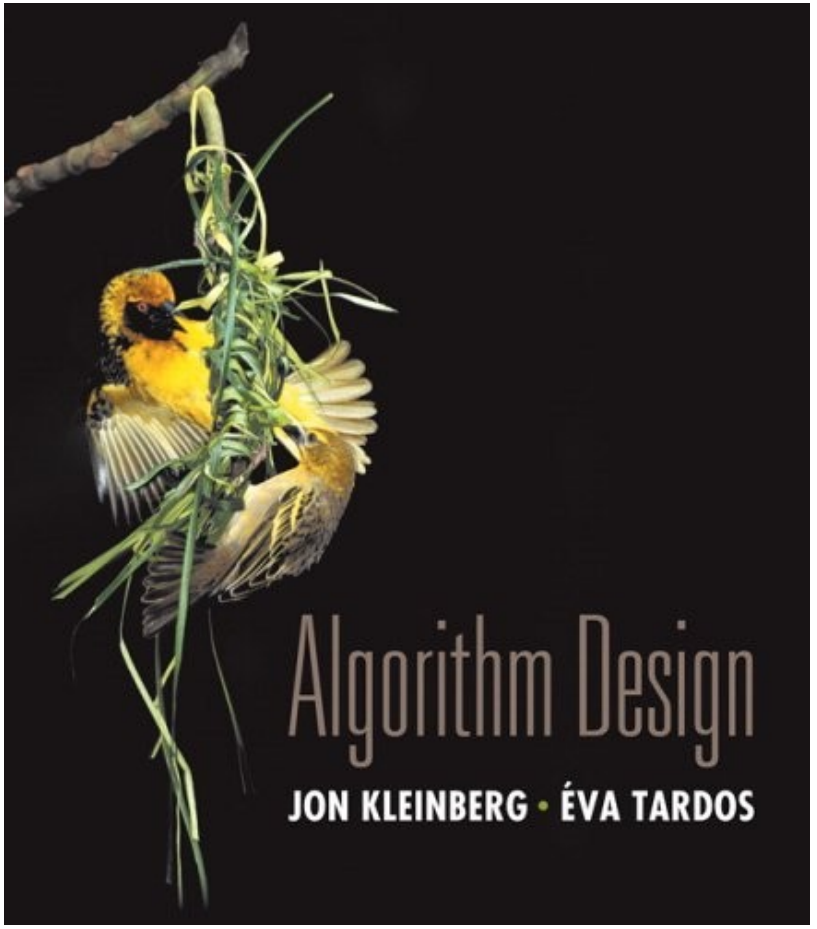
Andy Nguyen (tanonev@stanford.edu)
Julie Tibshirani (jtibs@cs.stanford.edu)
Kostas Kollias (kkollias@stanford.edu)
Phillip Chen (pcchen@stanford.edu)
Sean Choi (yo2seol@stanford.edu)

Course Staff Mailing List:
cs161-sum1213-staff@lists.stanford.edu

The Course Website

http://cs161.stanford.edu

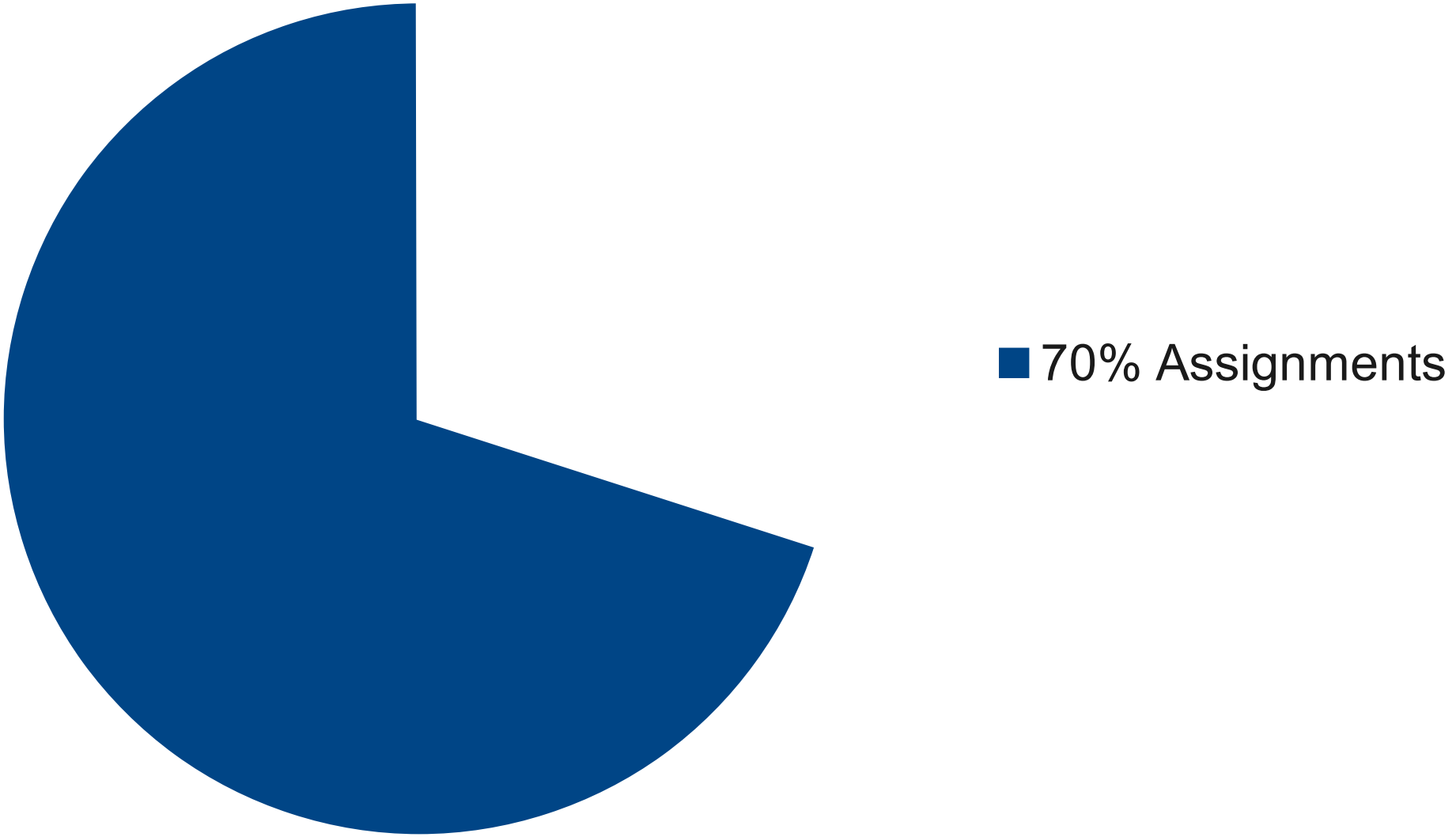
Required Reading



- *Algorithm Design* by Kleinberg and Tardos.
- Available in the Stanford Bookstore.

Grading Policies

Grading Policies

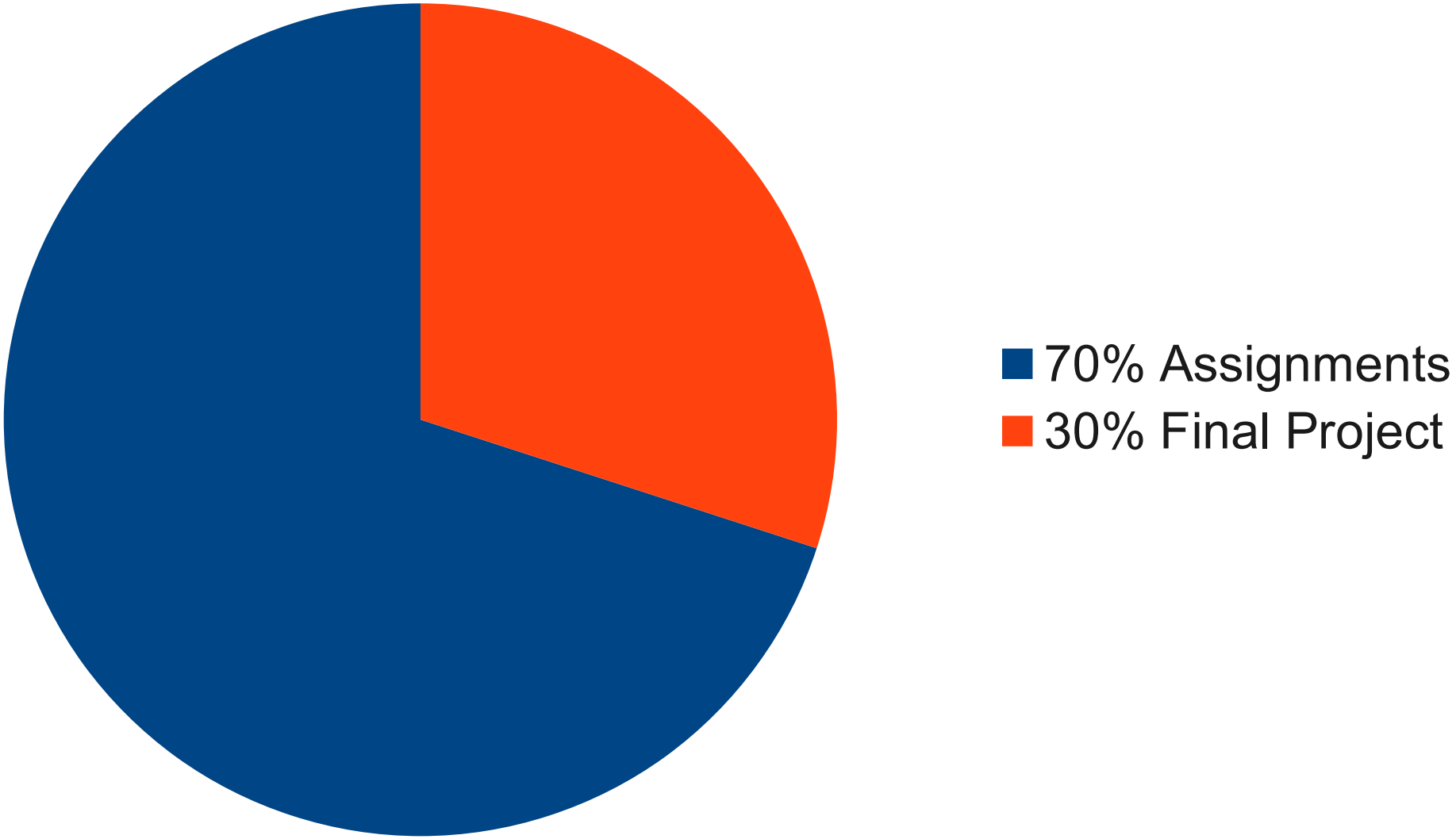


Grading Policies

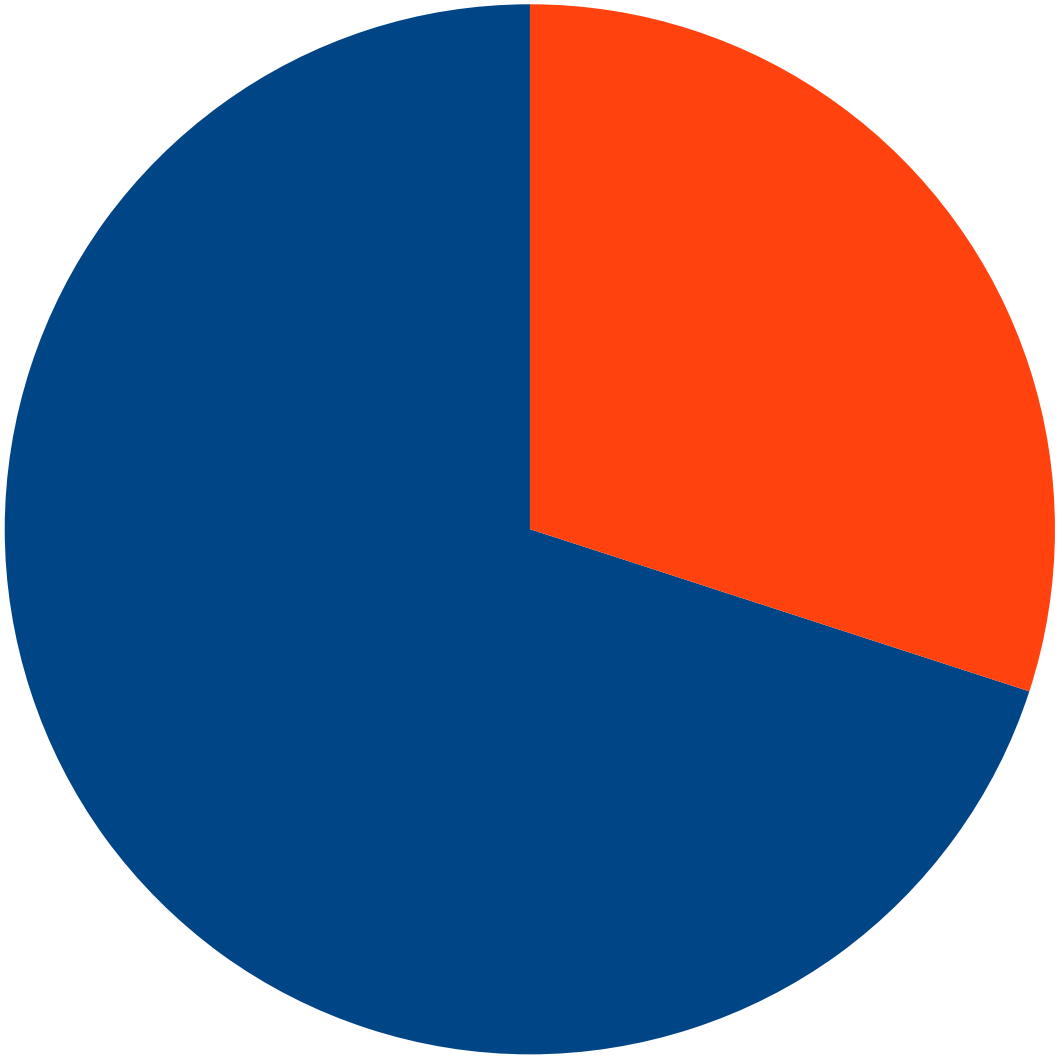
70% Assignments

Six Written Assignments

Grading Policies



Grading Policies



70% Assignments

30% Final Project

Final Project due Saturday, August 17 at 12:15PM

Prerequisites

- **CS103** (Mathematical Foundations of Computing)
  - Mathematical proof.
  - Discrete math (sets, functions, graphs, relations.)
  - Undecidability
  - **P** $\stackrel{?}{=}$ **NP**
  - (etc.)

Prerequisites

- **CS109** (Probability for Computer Scientists)
  - Linearity of expectation
  - Random variables
  - Conditional probability
  - (etc.)

Prerequisites

- **CS106B/X** (Programming Abstractions)
  - Recursion and recursive problem solving.
  - Fundamental data structures (stacks, queues, lists, binary search trees, hash tables.)
  - Basic algorithmic analysis using big-O notation.
  - (etc.)

Why Study Algorithms?

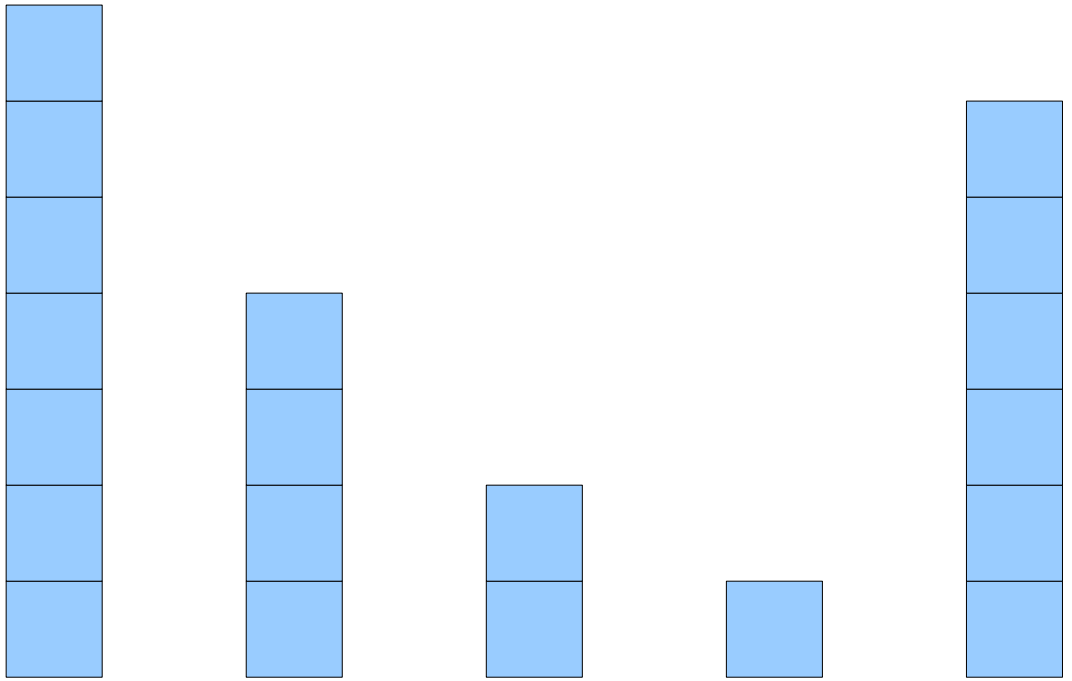
Key Questions

- **How do you find efficient solutions to seemingly hard problems?**
- **How do you prove that an algorithm is correct?**
- **How do you analyze an algorithm's runtime?**

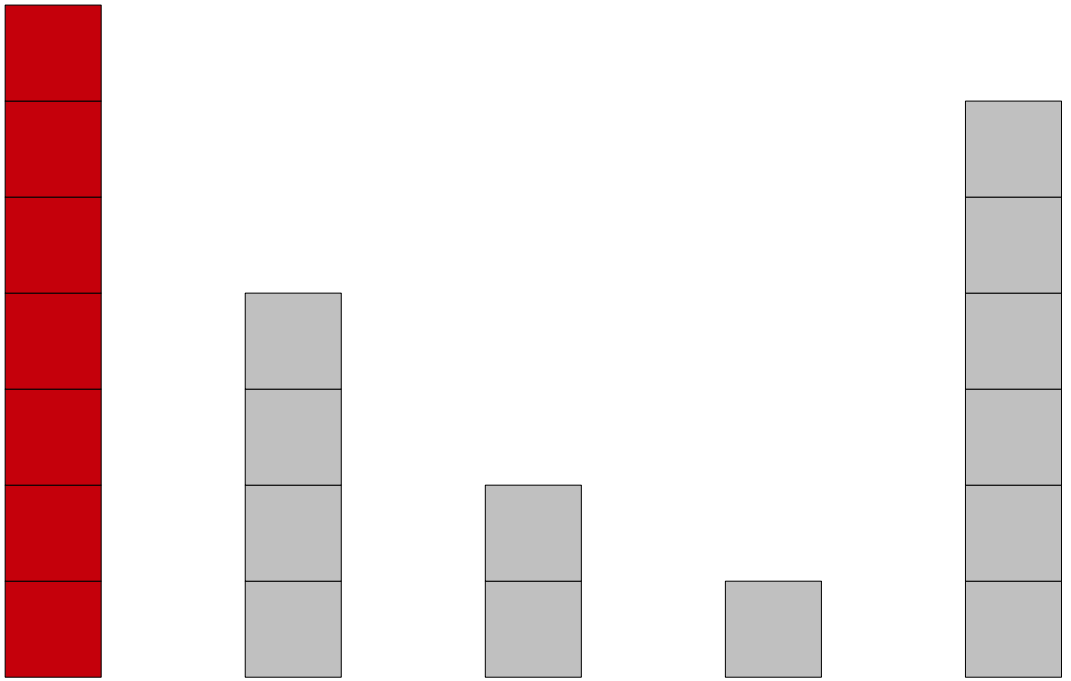
A First Algorithm: **Insertion Sort**

Insertion Sort

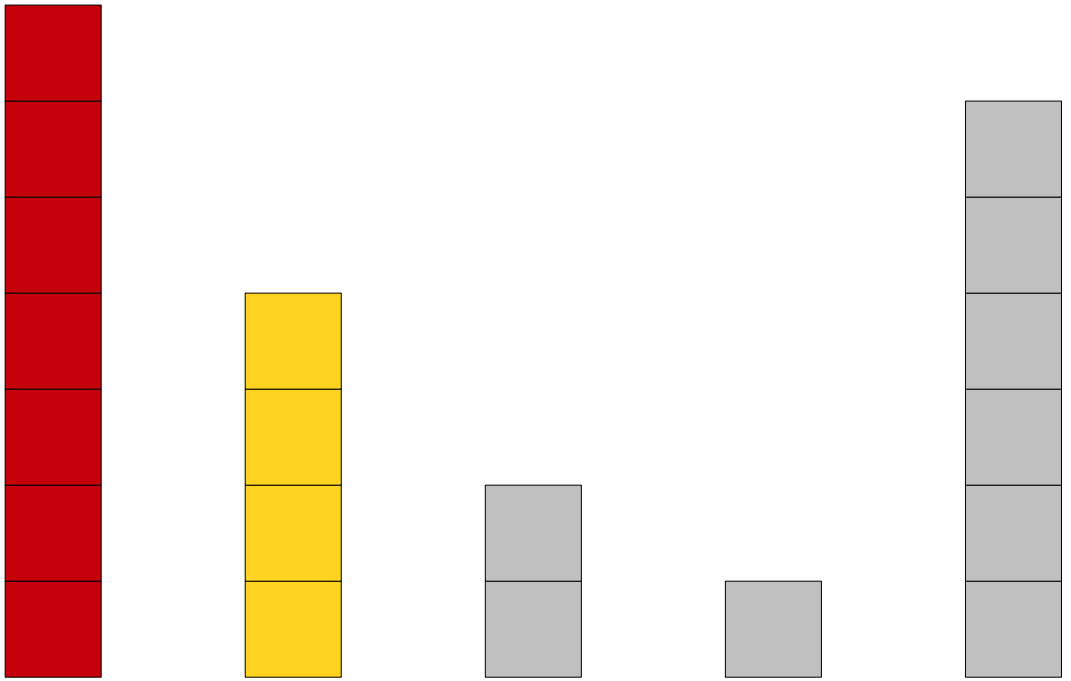
Insertion Sort



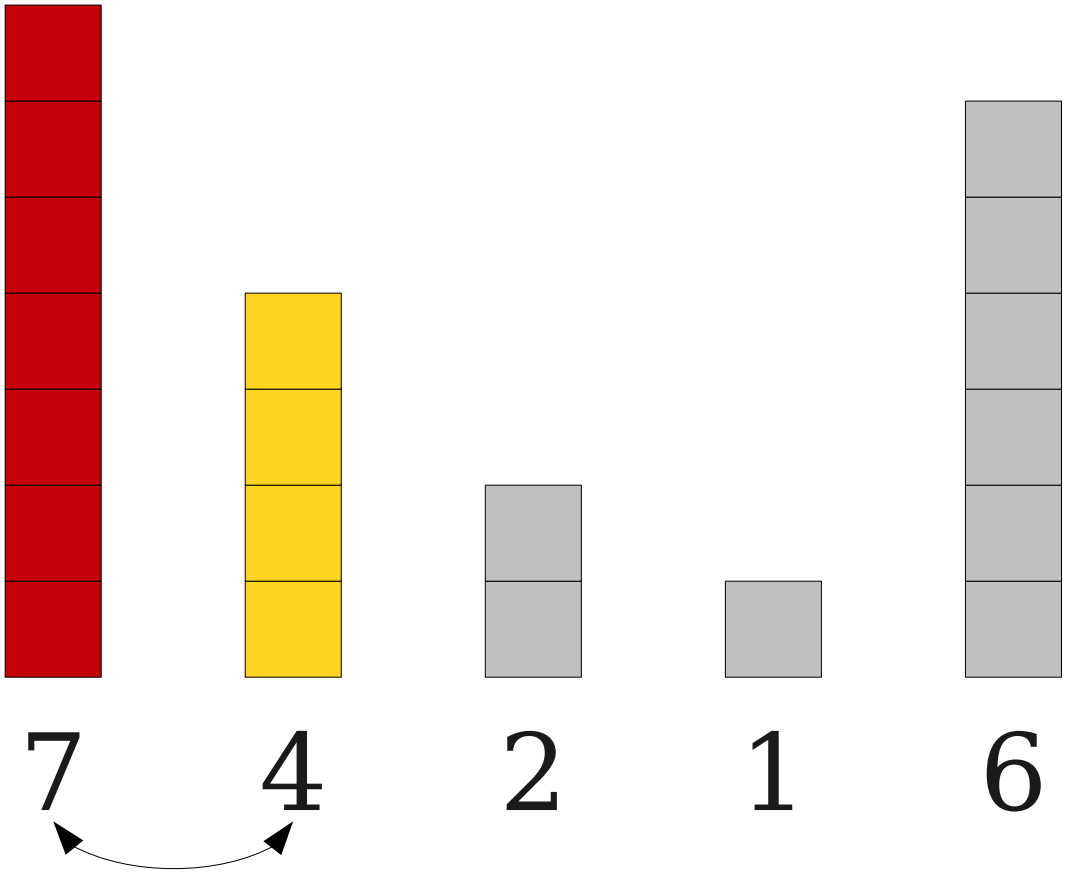
Insertion Sort



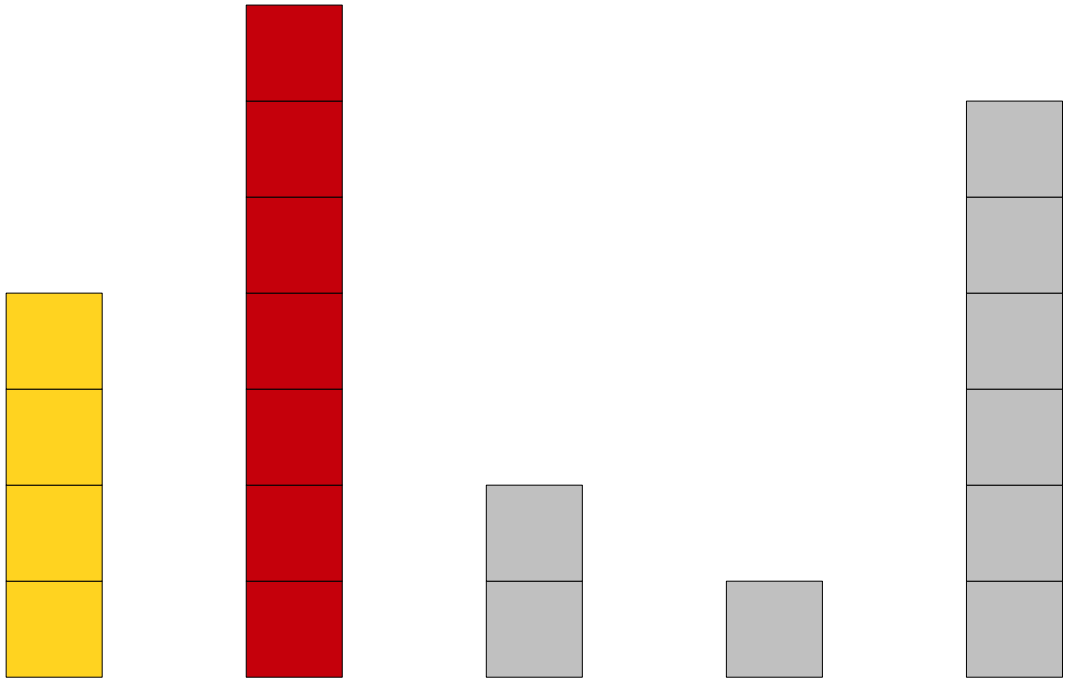
Insertion Sort



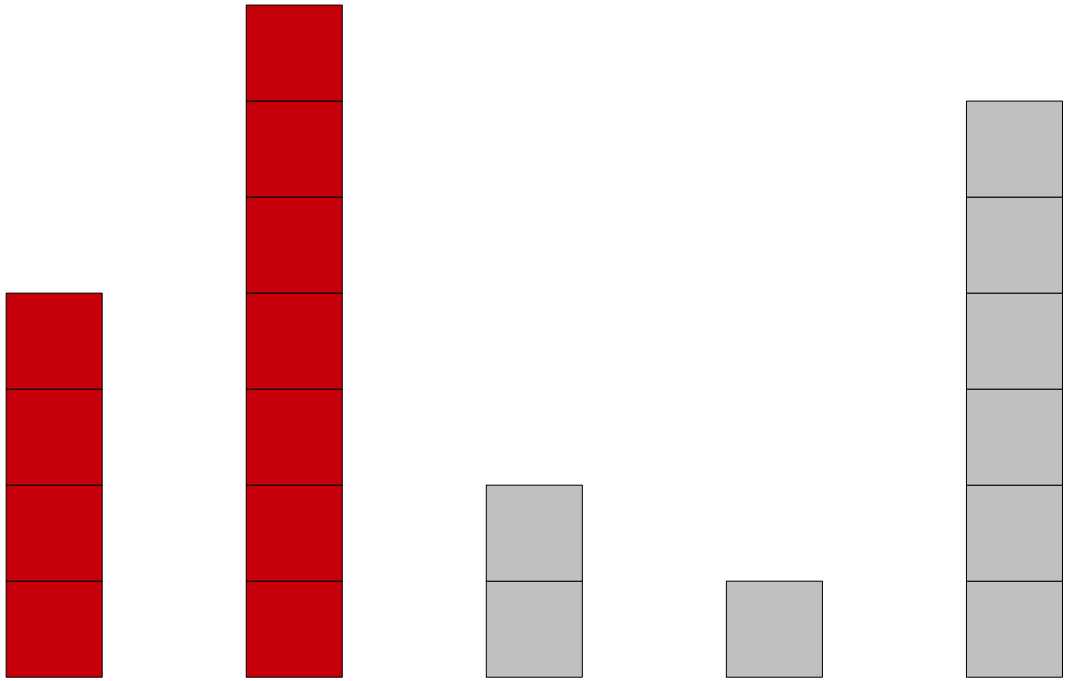
Insertion Sort



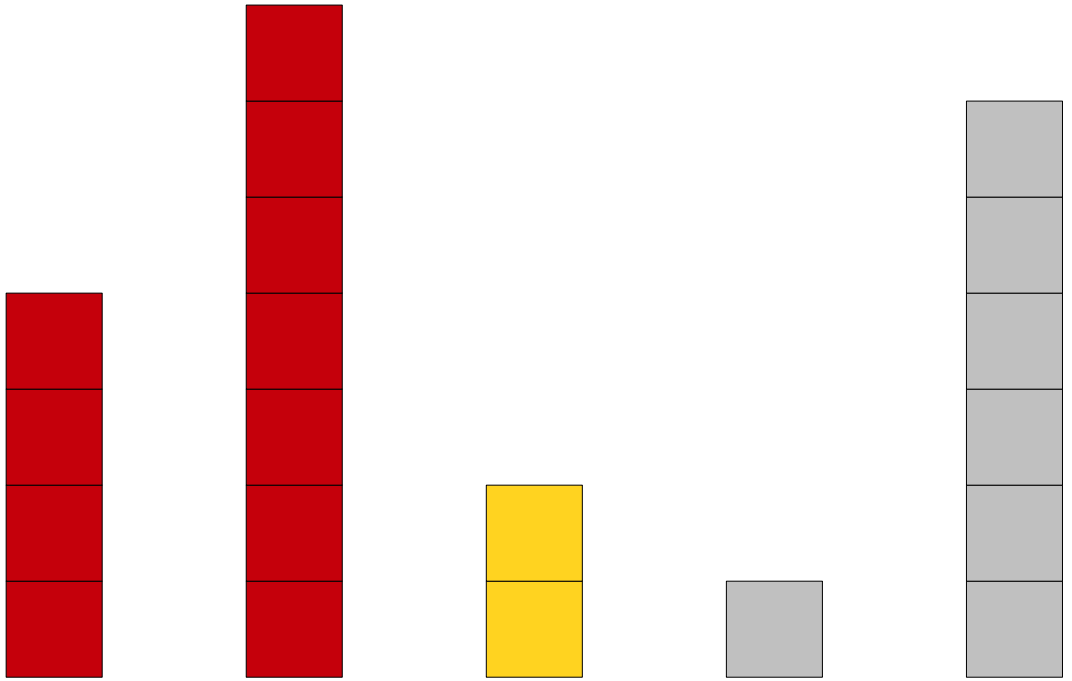
Insertion Sort



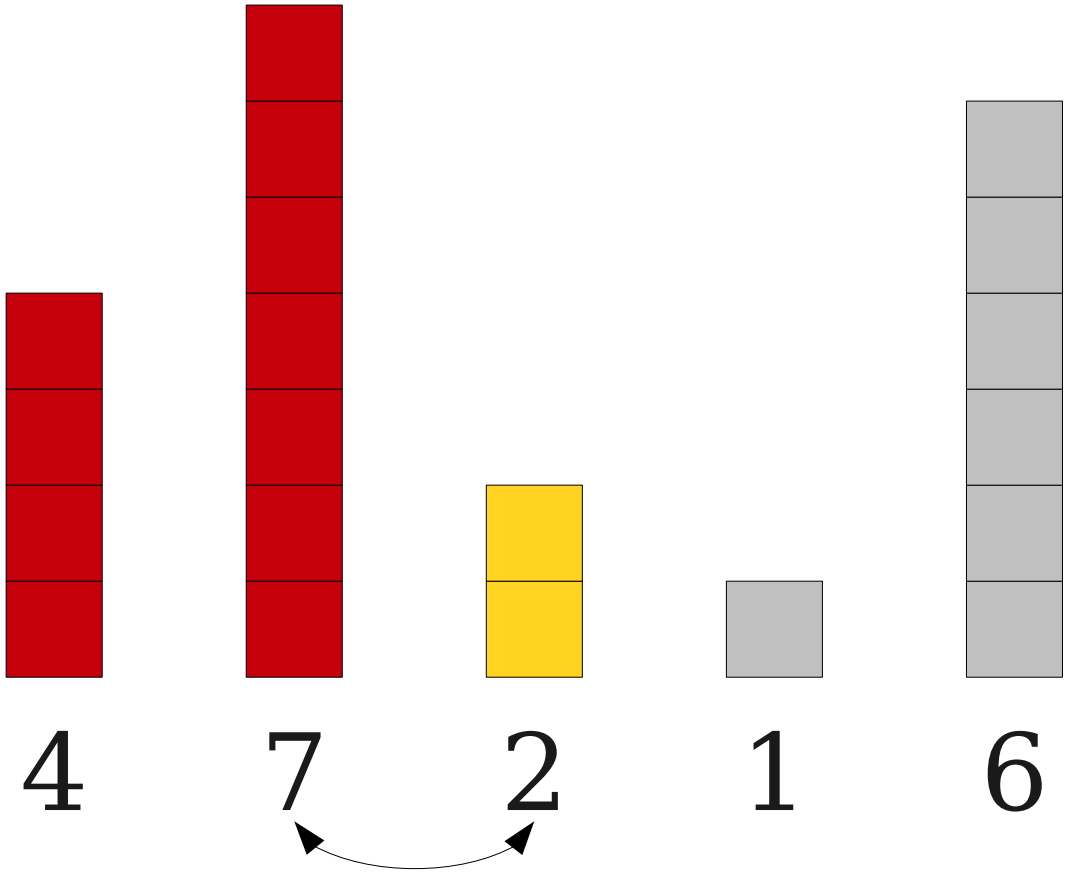
Insertion Sort



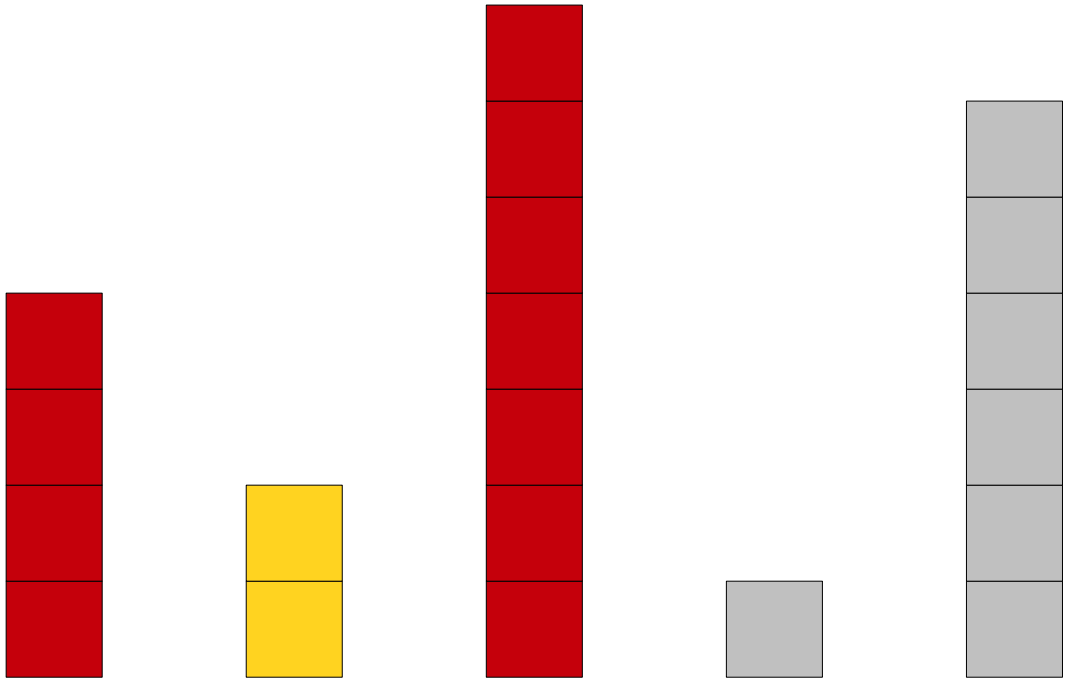
Insertion Sort



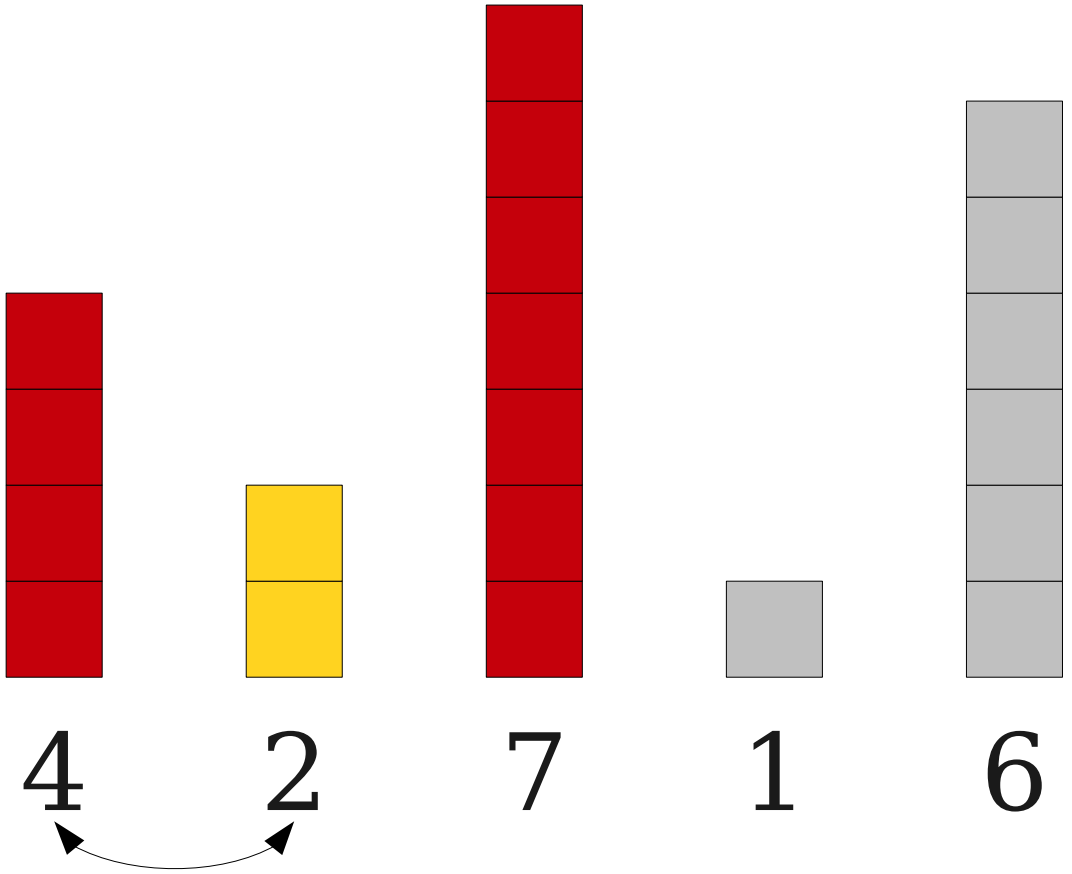
Insertion Sort



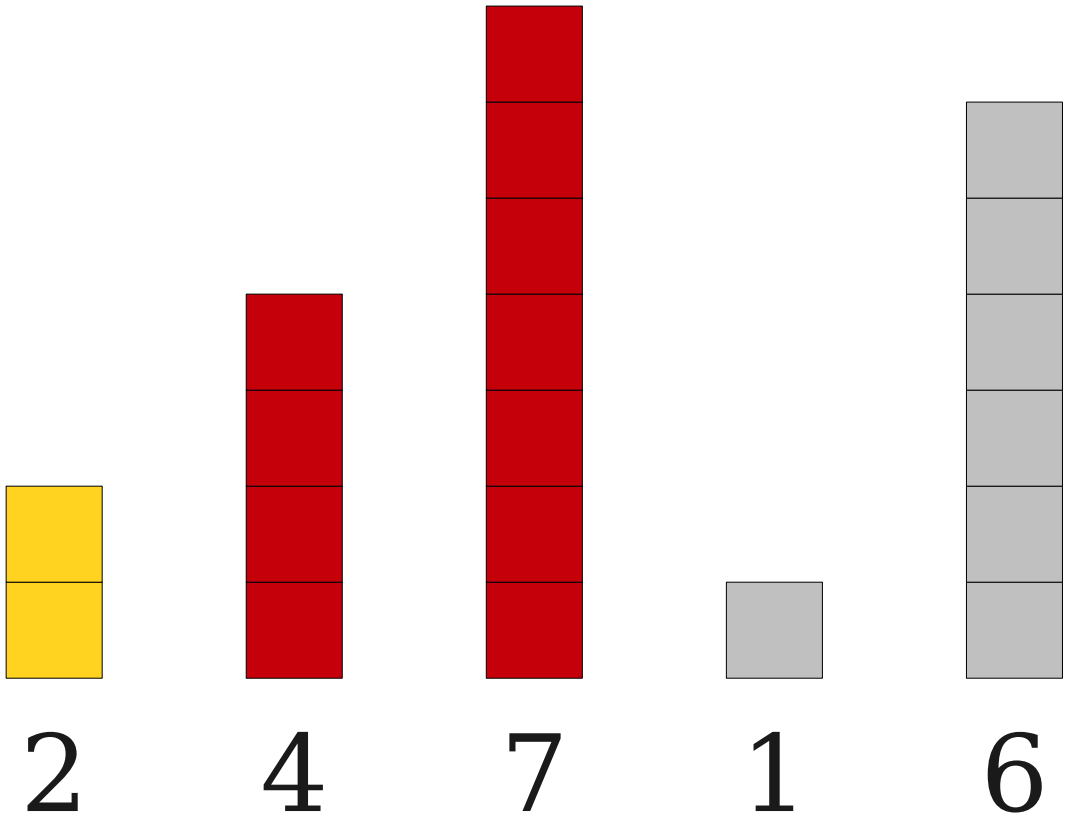
Insertion Sort



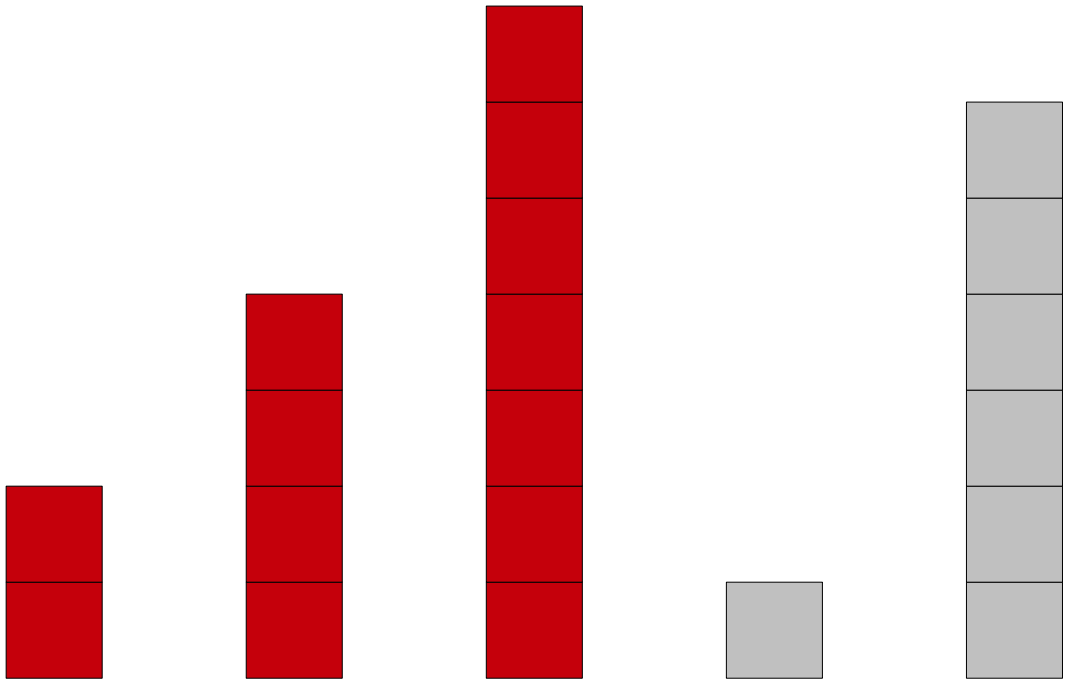
Insertion Sort



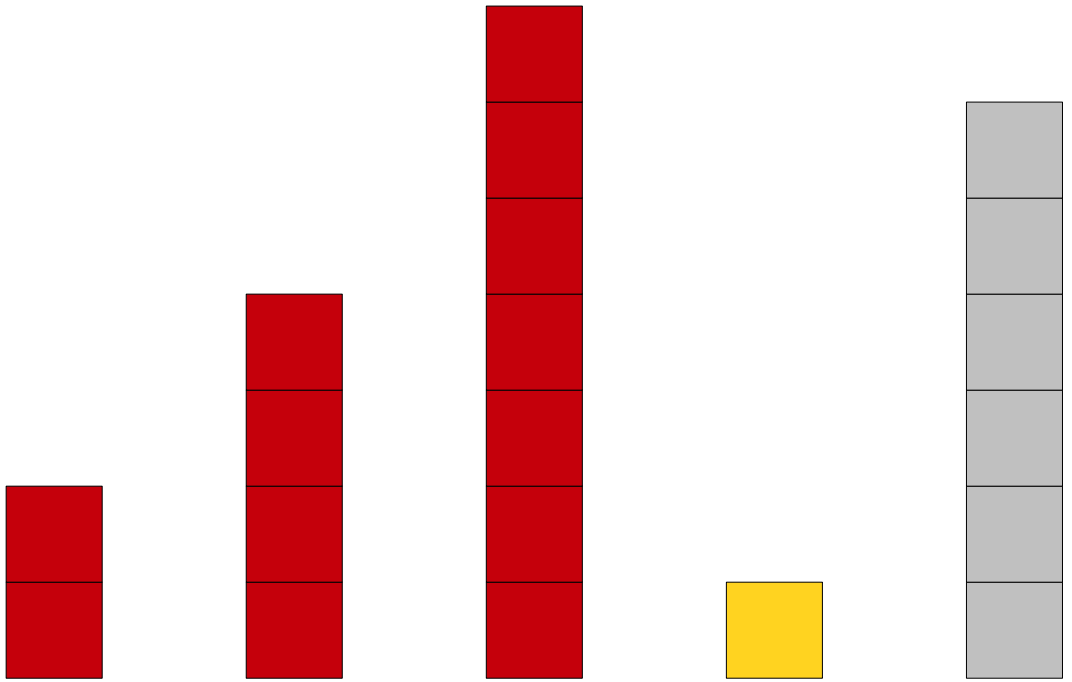
Insertion Sort



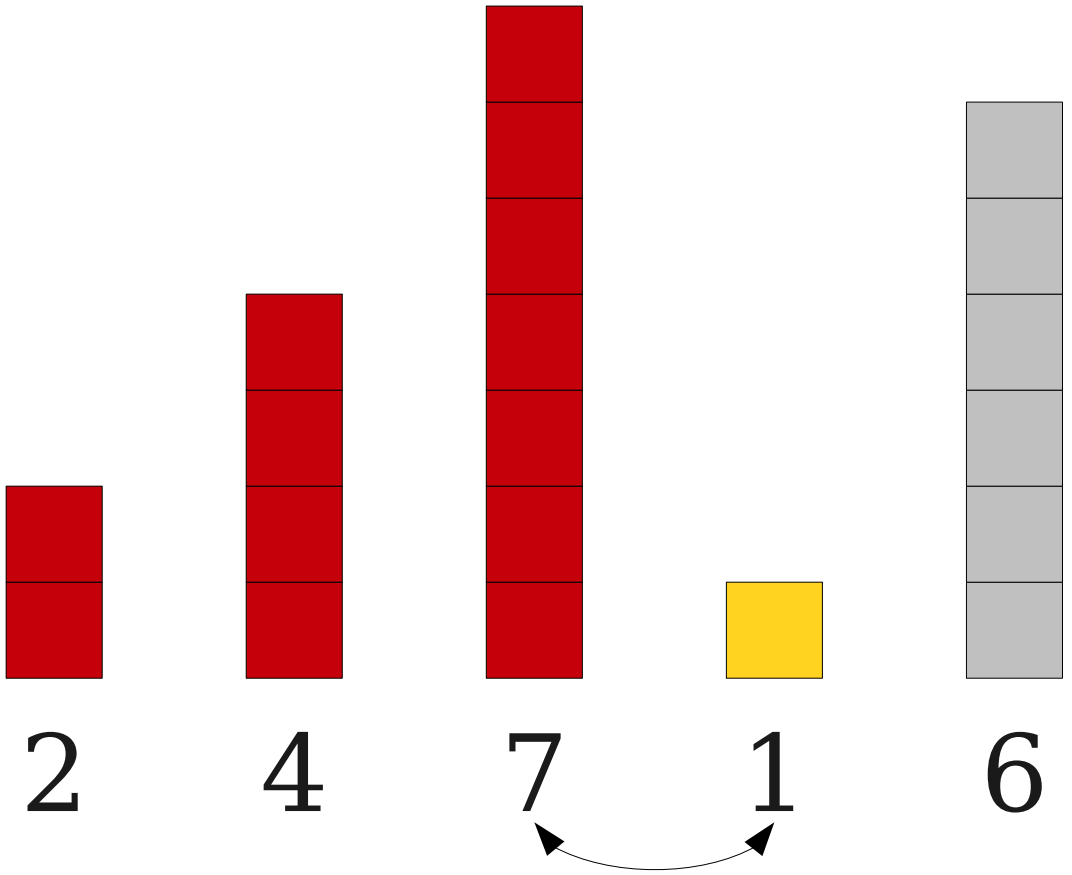
Insertion Sort



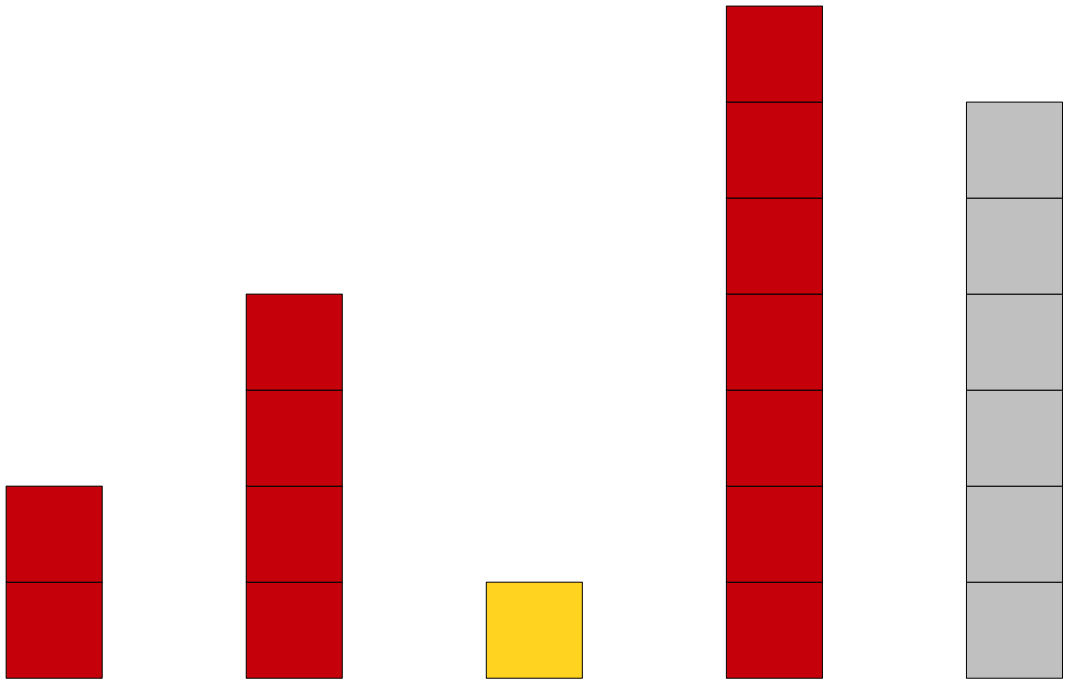
Insertion Sort



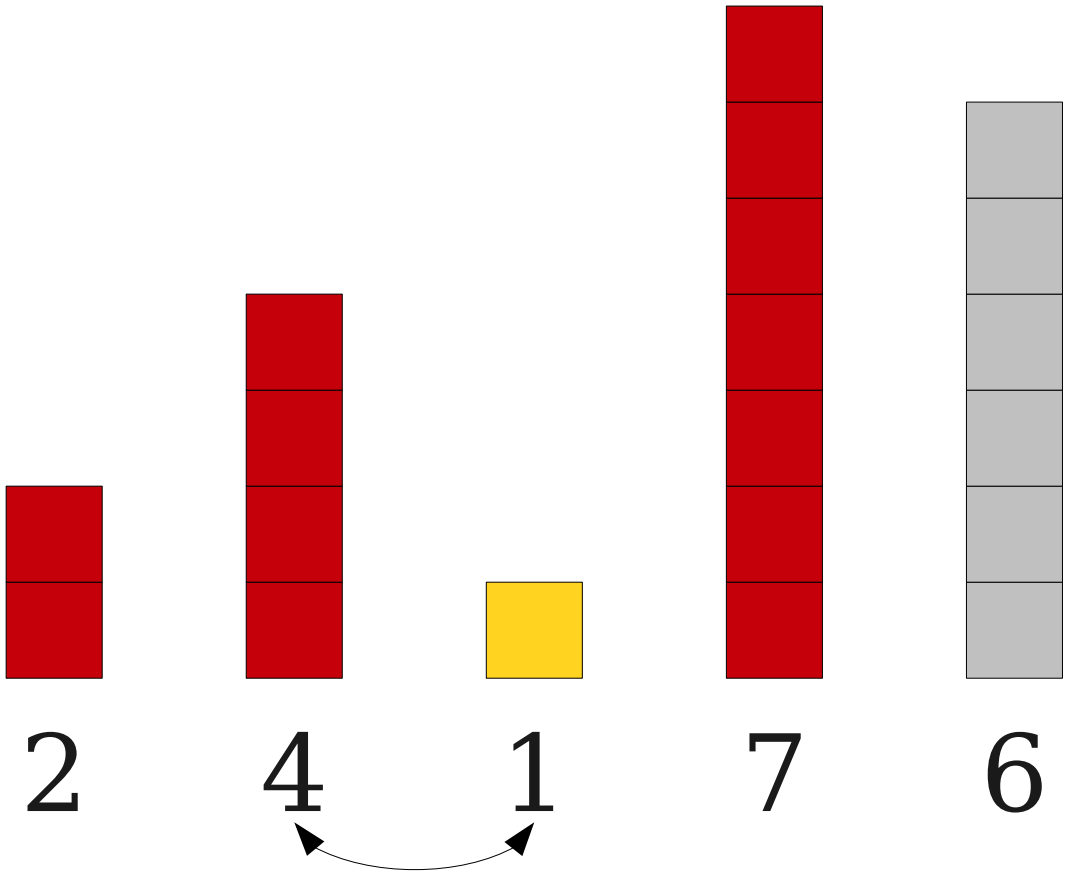
Insertion Sort



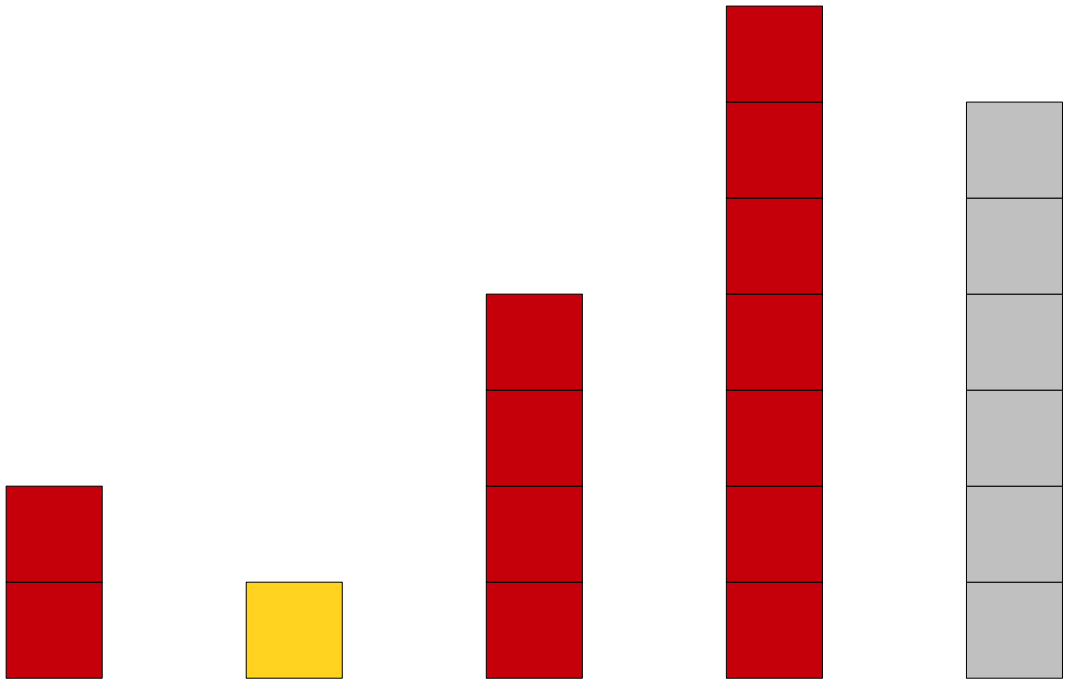
Insertion Sort



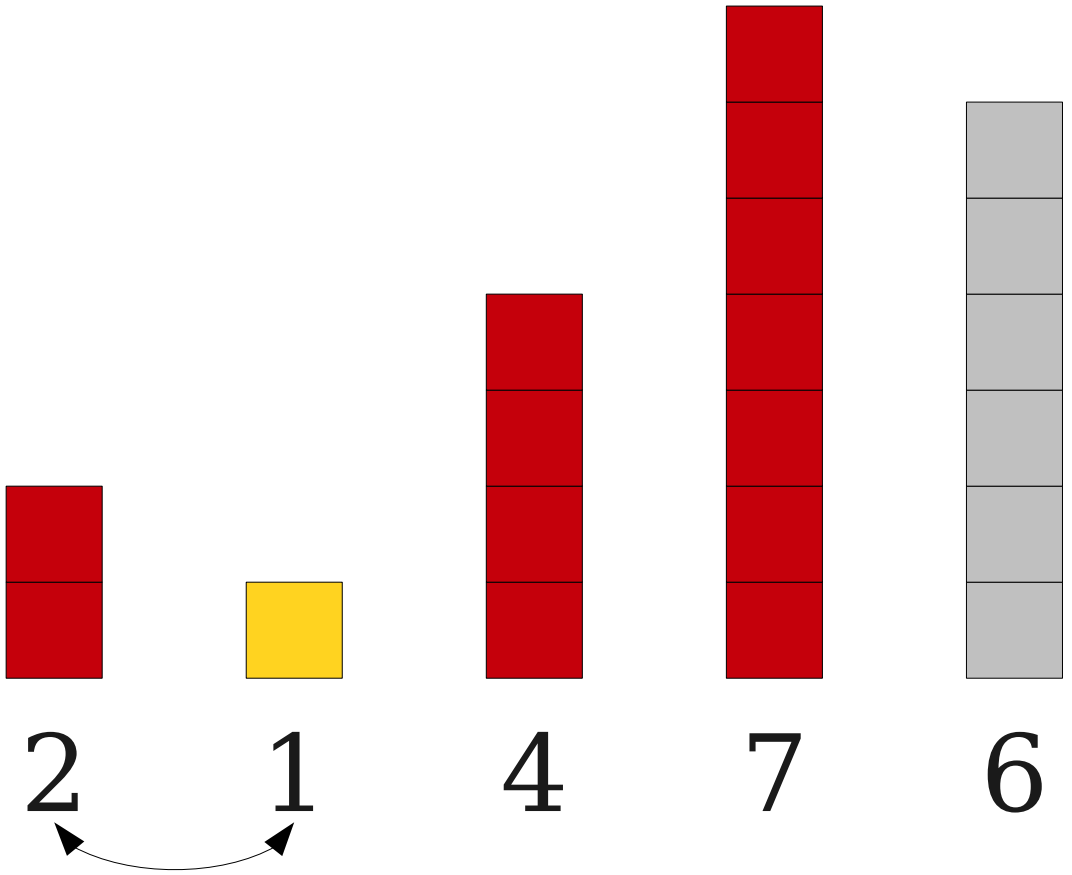
Insertion Sort



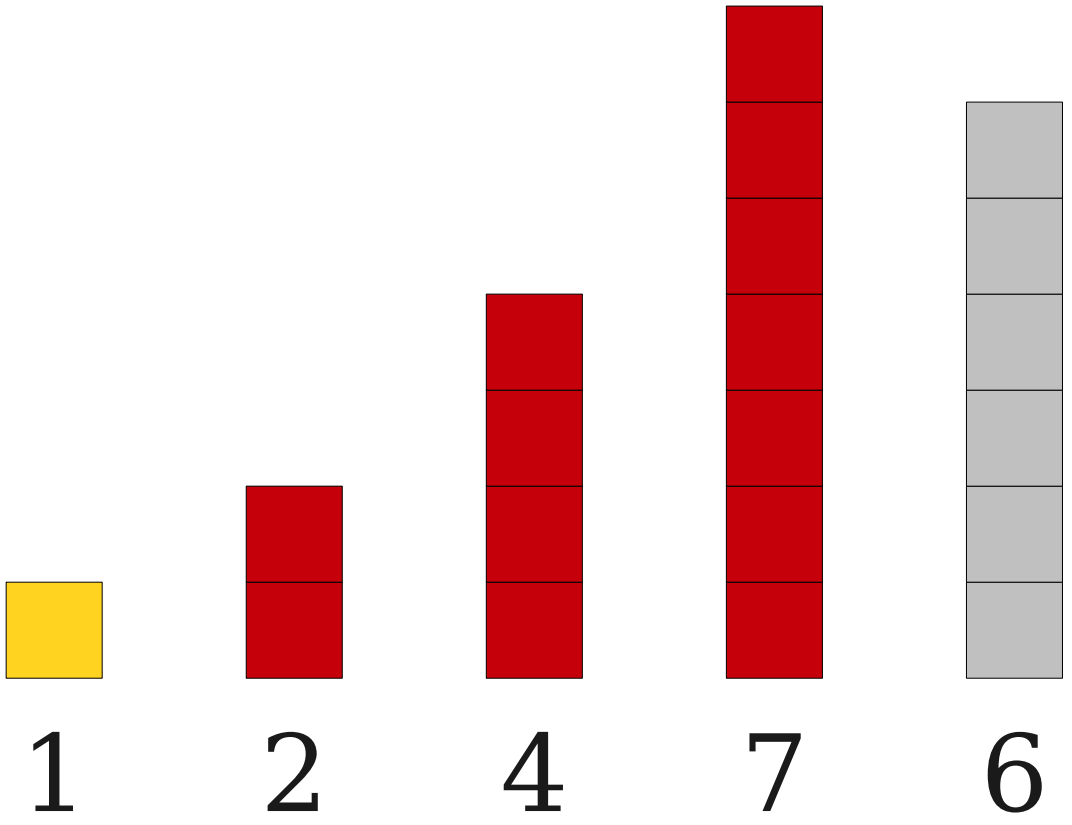
Insertion Sort



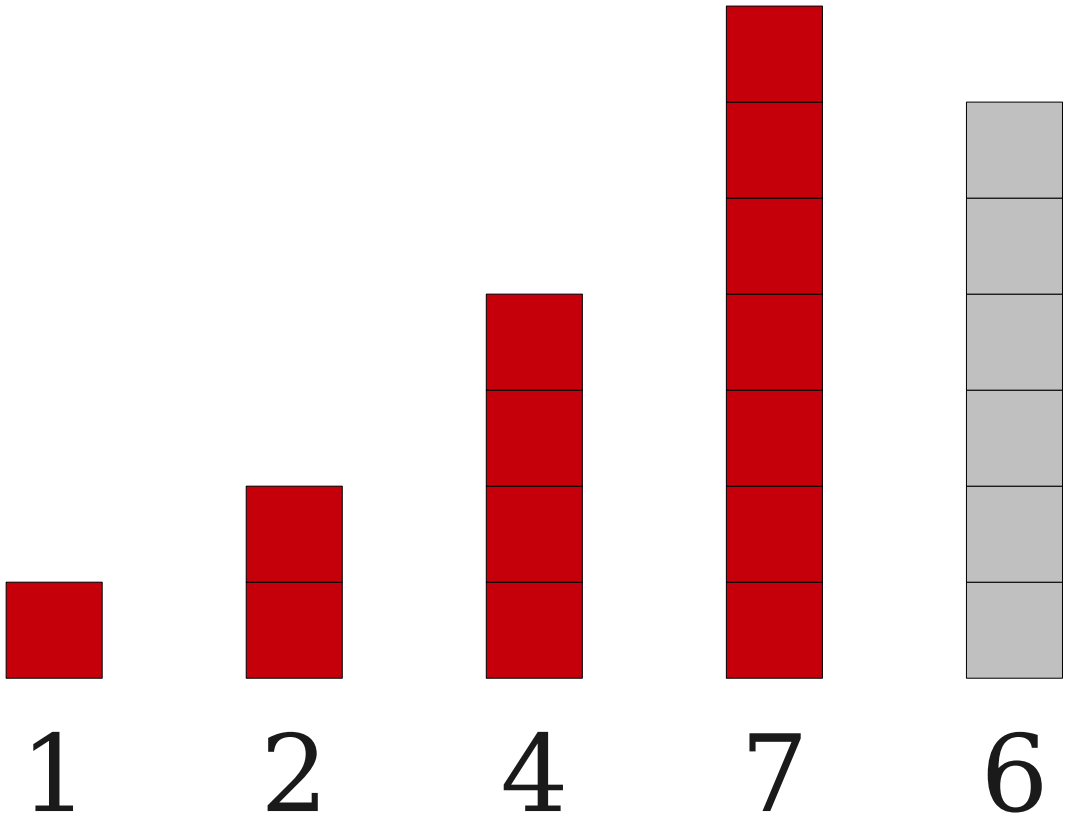
Insertion Sort



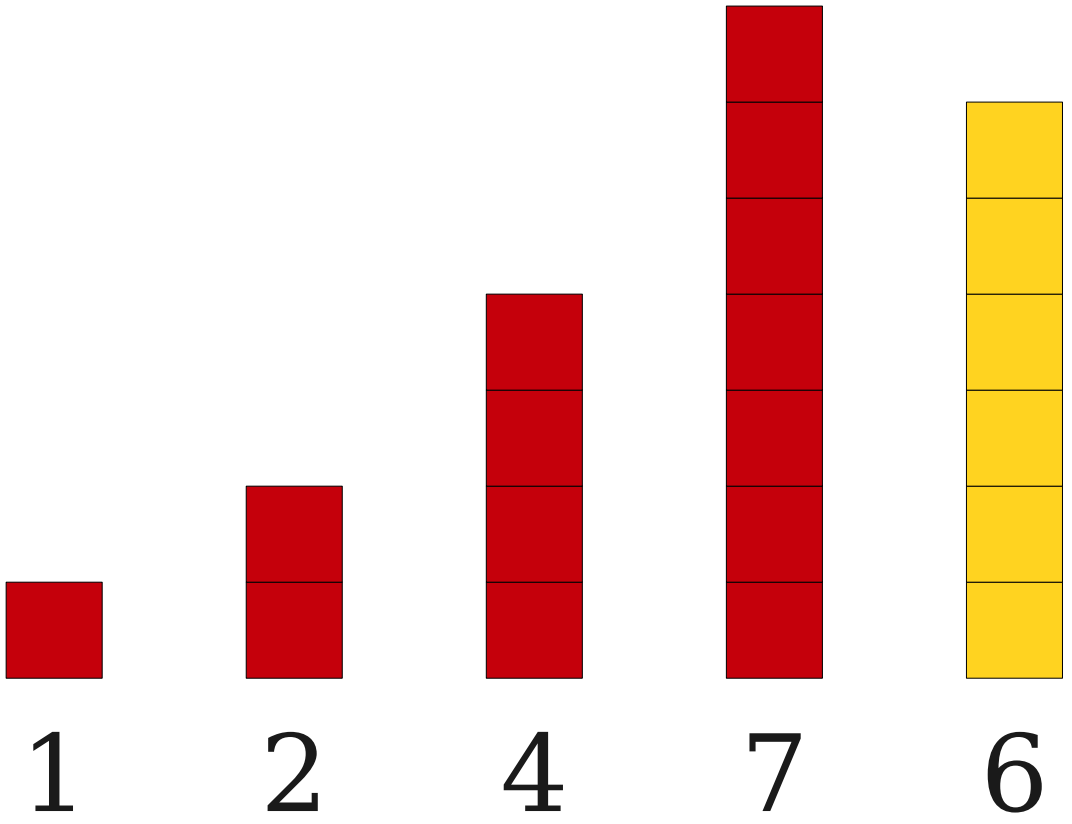
Insertion Sort



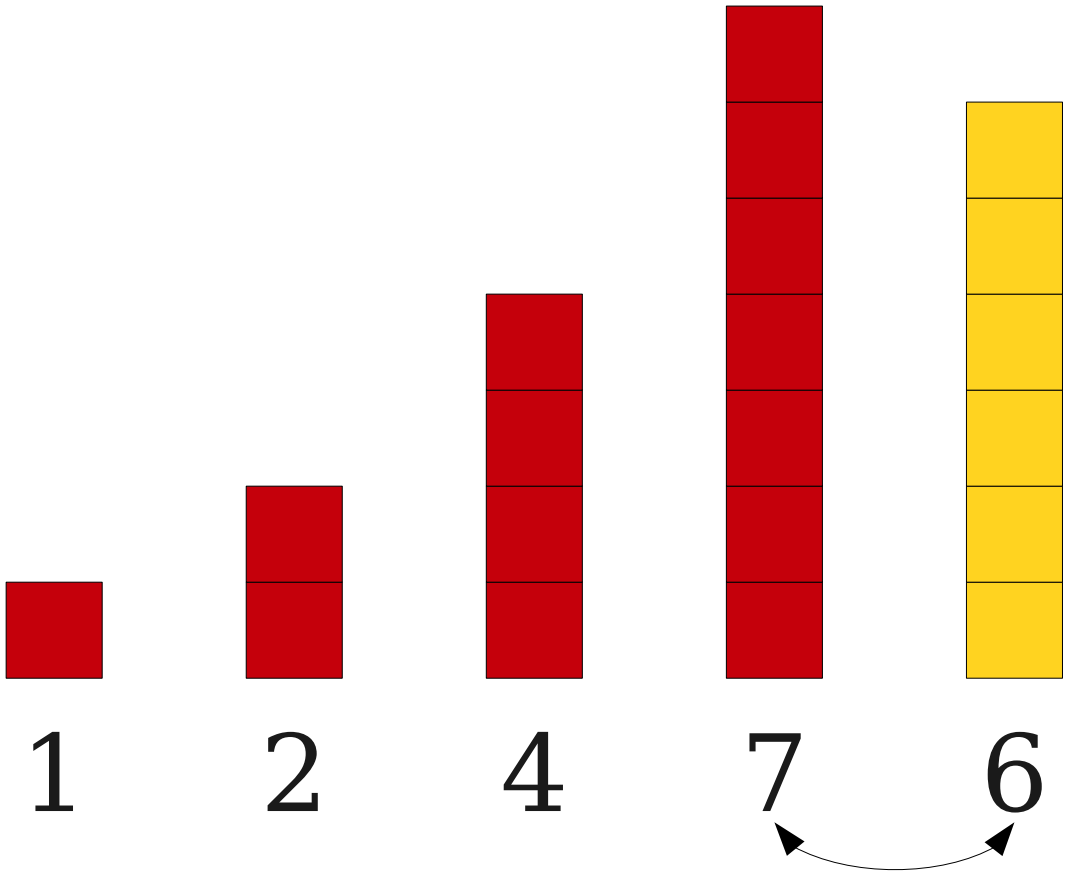
Insertion Sort



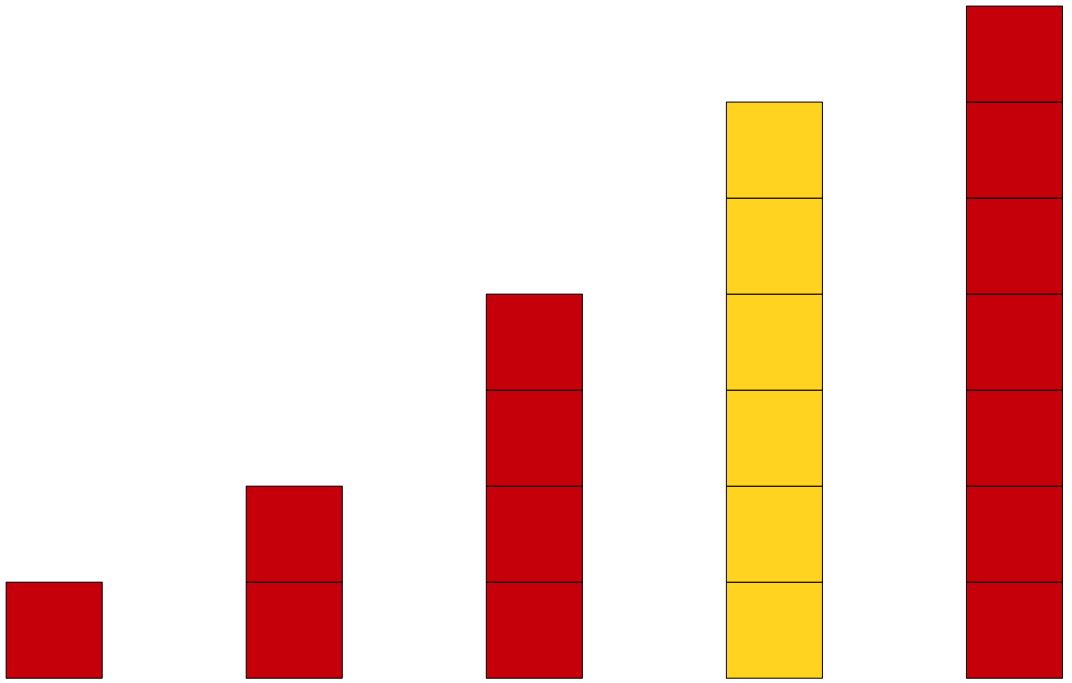
Insertion Sort



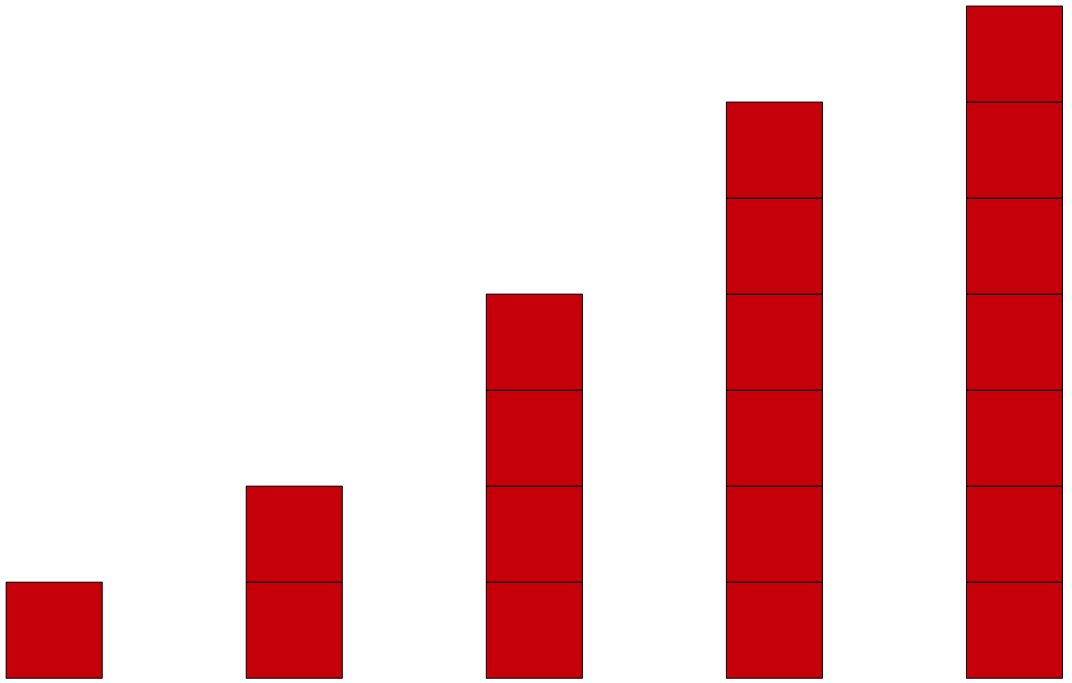
Insertion Sort



Insertion Sort



Insertion Sort



```
procedure insertionSort(list A):
   for i = 0 to length(A) - 1
      let j = i
      while j > 0 and A[j - 1] > A[j]:
         swap A[j - 1] and A[j]
         j = j - 1
```

Question 1: How do we prove this always sorts the input array?

Question 2: How *efficiently* does insertion sort sort the input array?

Question 1: How do we prove this always sorts the input array?

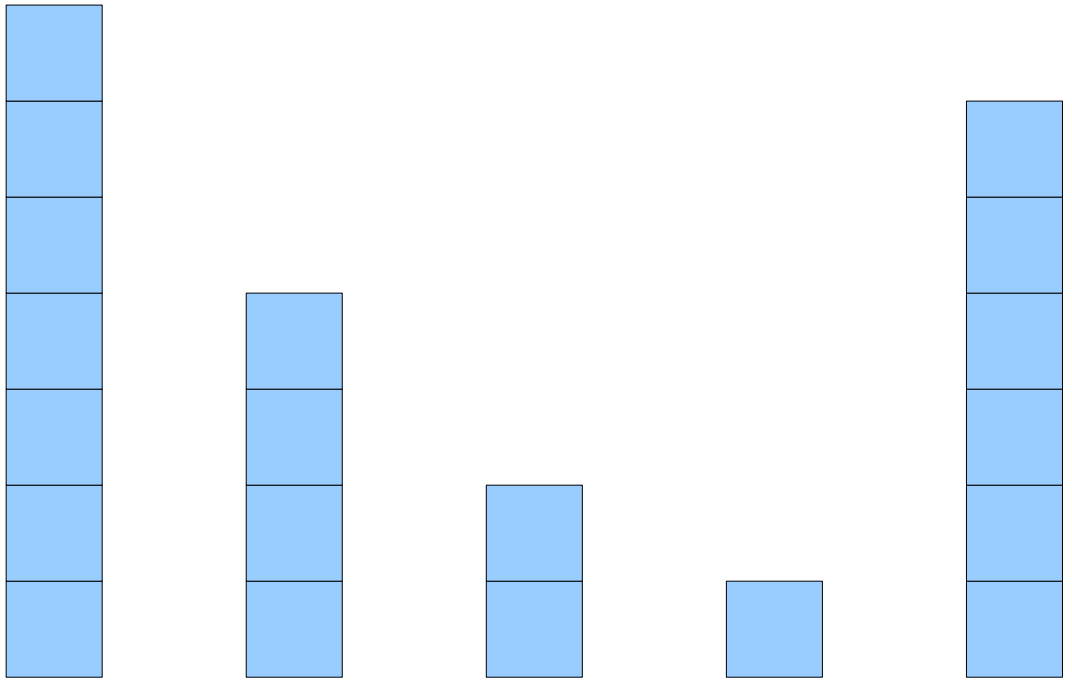
Question 2: How *efficiently* does insertion sort sort the input array?

Proving Correctness

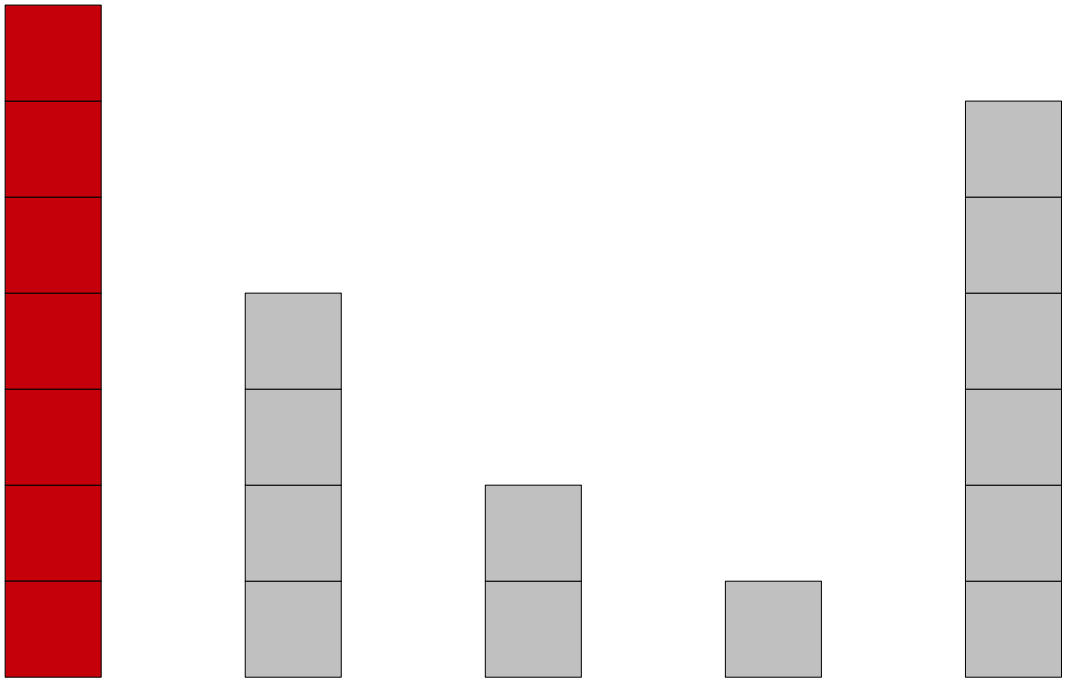
- Algorithms represent a process during which data is created, modified, or destroyed.
- Reasoning about these processes can be tricky because so much data generated or changed.
- It often helps to look for things that *don't* change.
- An **invariant** is a property of data that is unmodified by an algorithm, typically a set of relationships that hold between elements.
- Finding invariants makes it easier to prove properties of the algorithm.

Insertion Sort

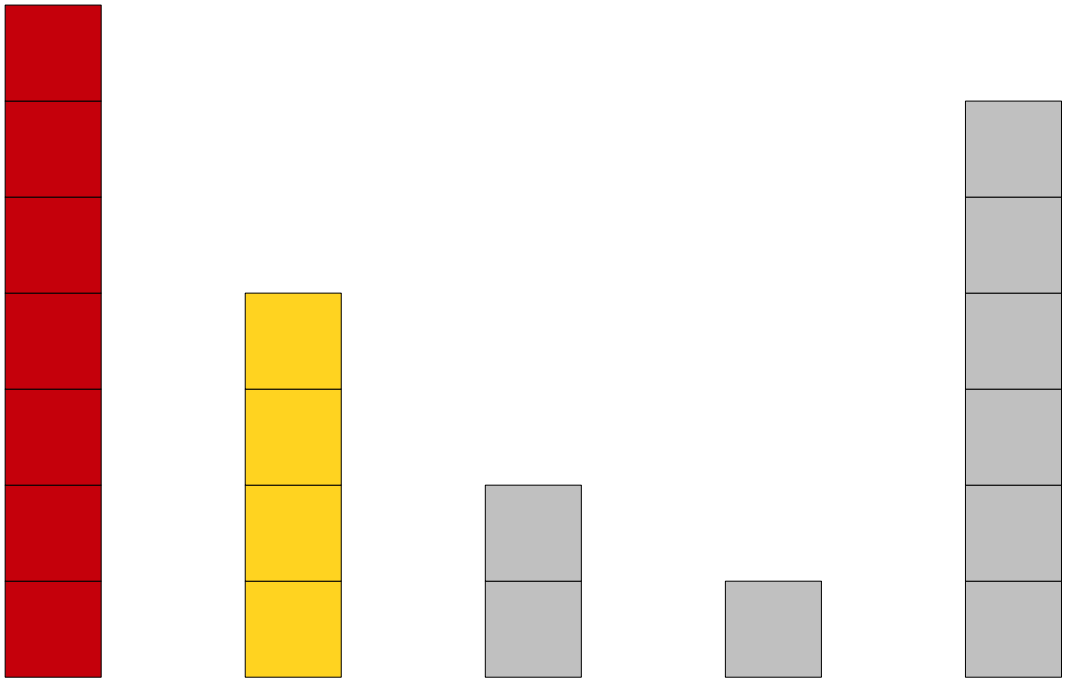
Insertion Sort



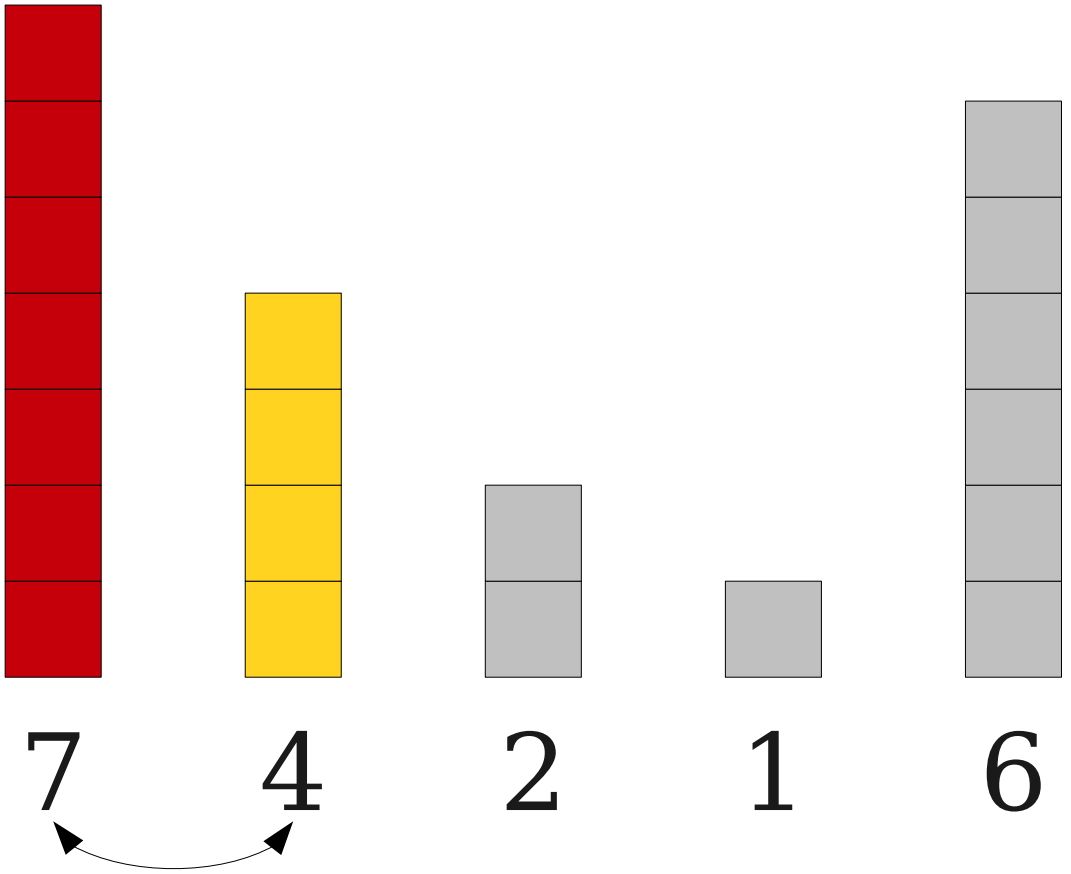
Insertion Sort



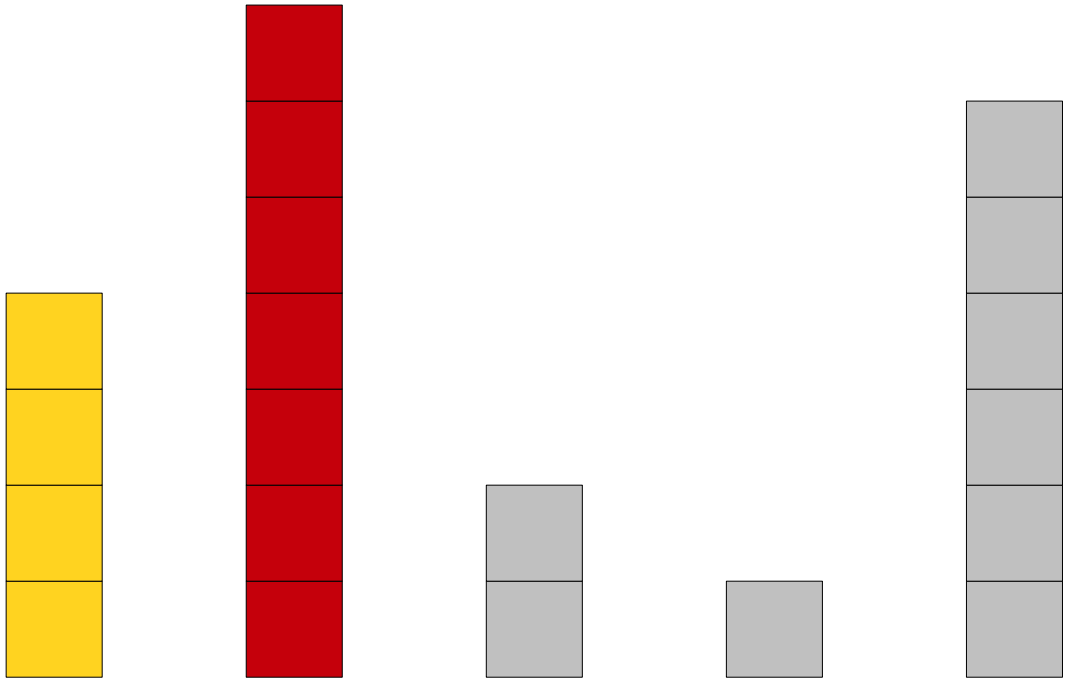
Insertion Sort



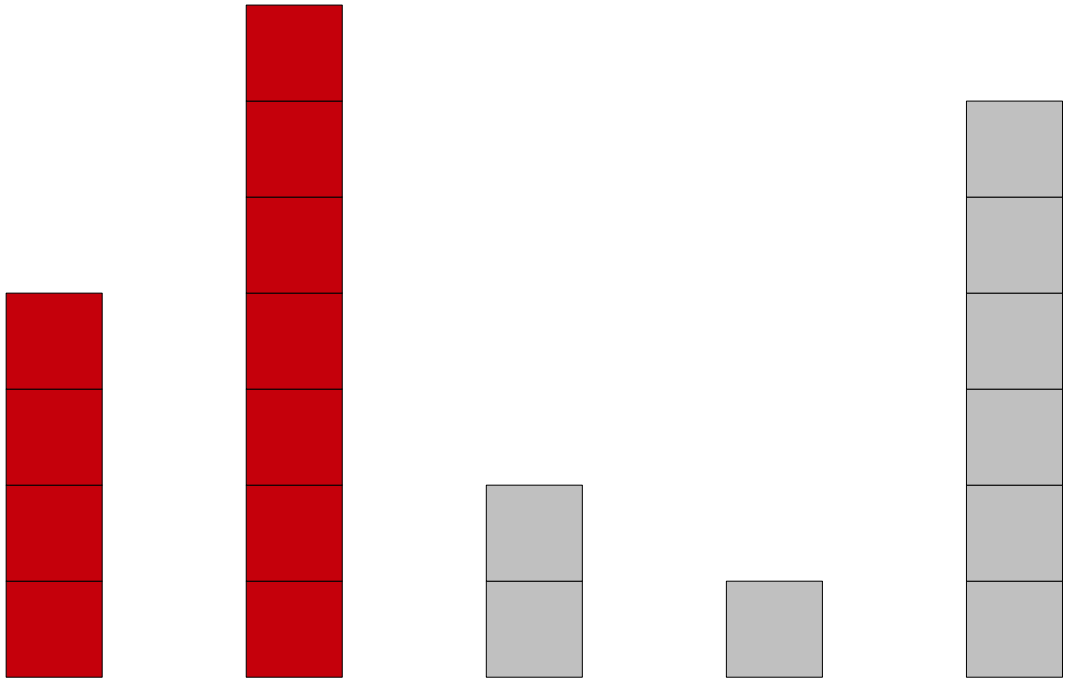
Insertion Sort



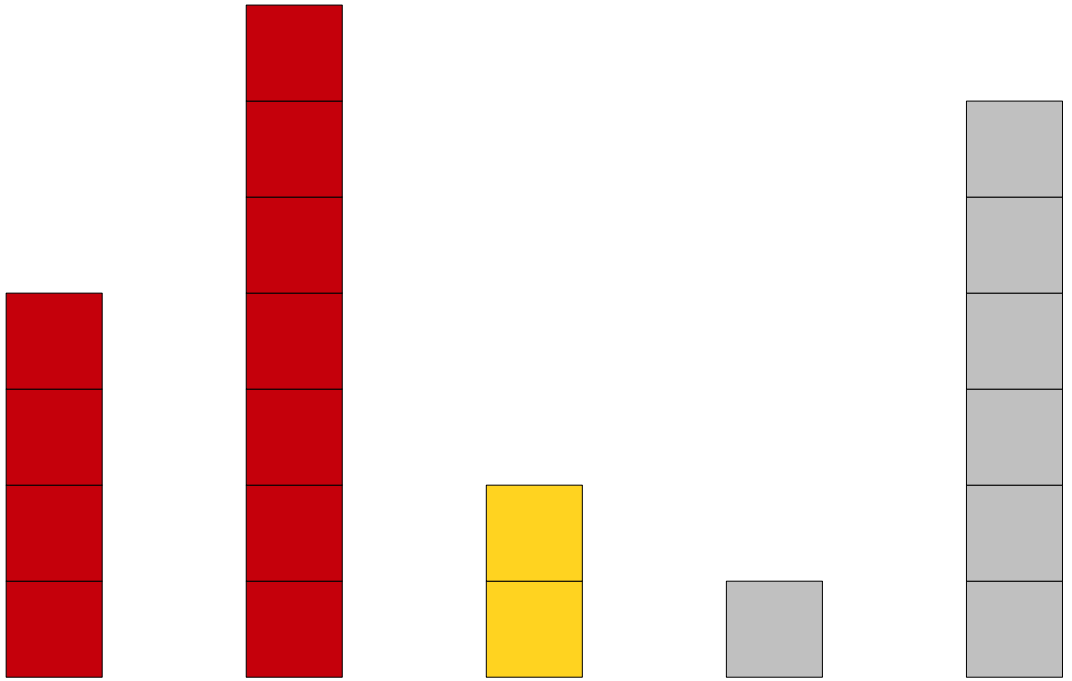
Insertion Sort



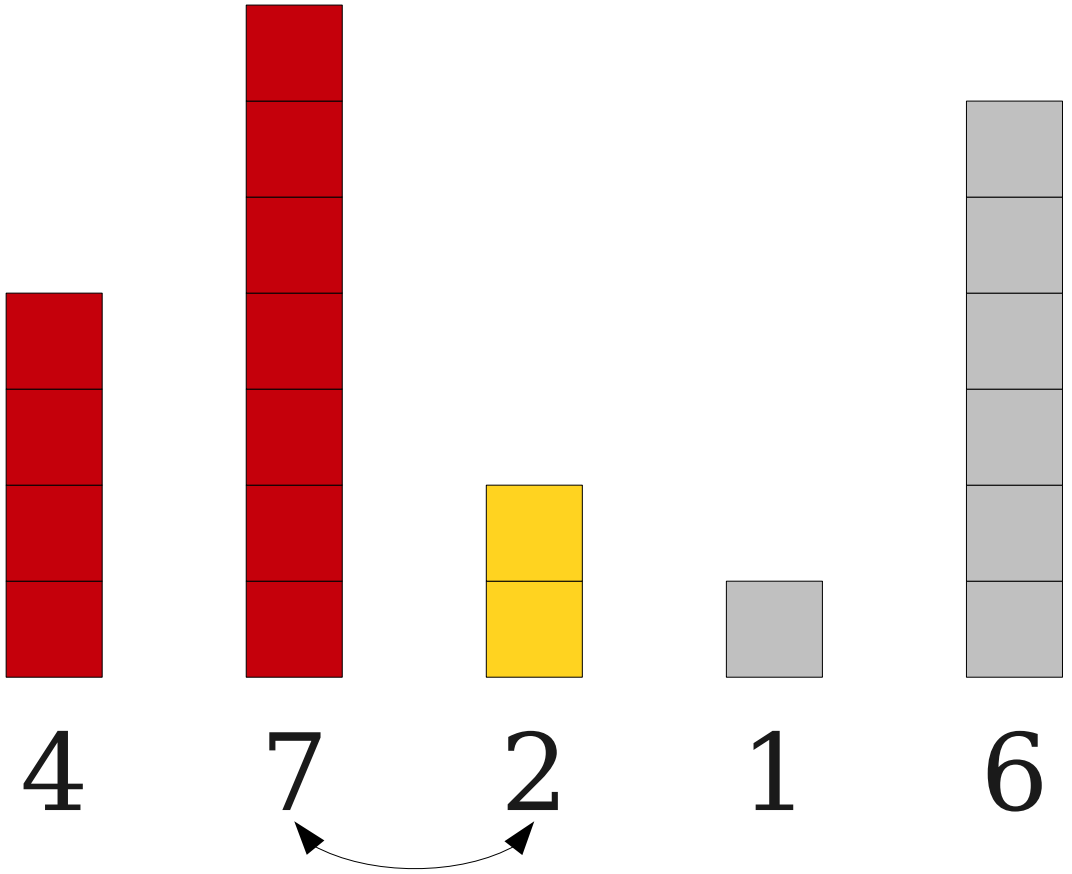
Insertion Sort



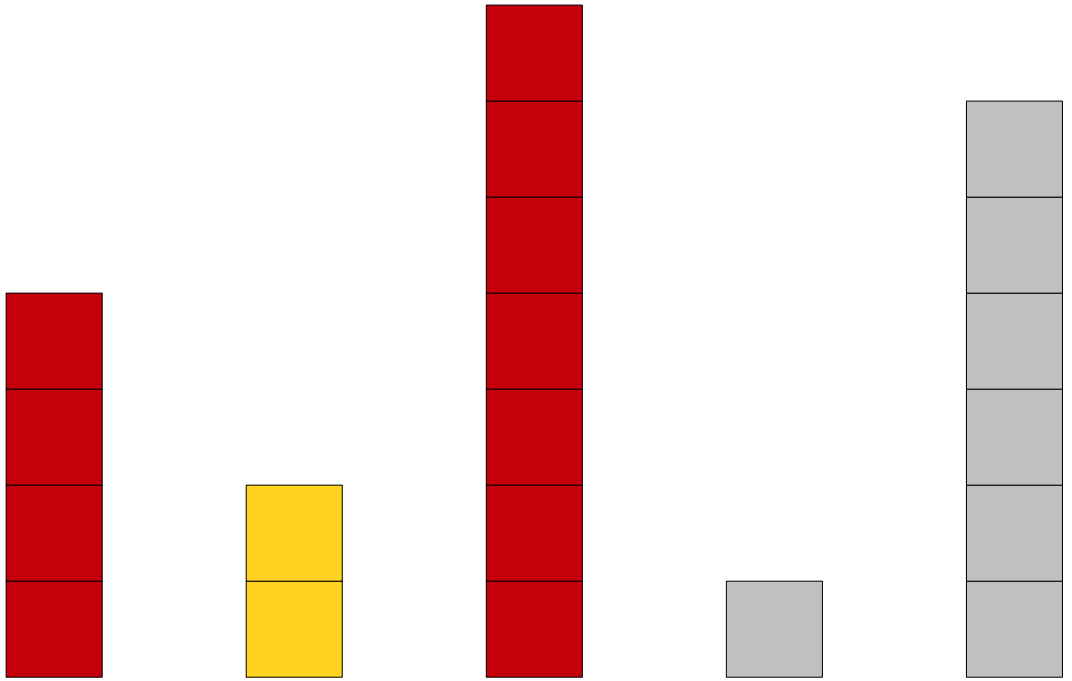
Insertion Sort



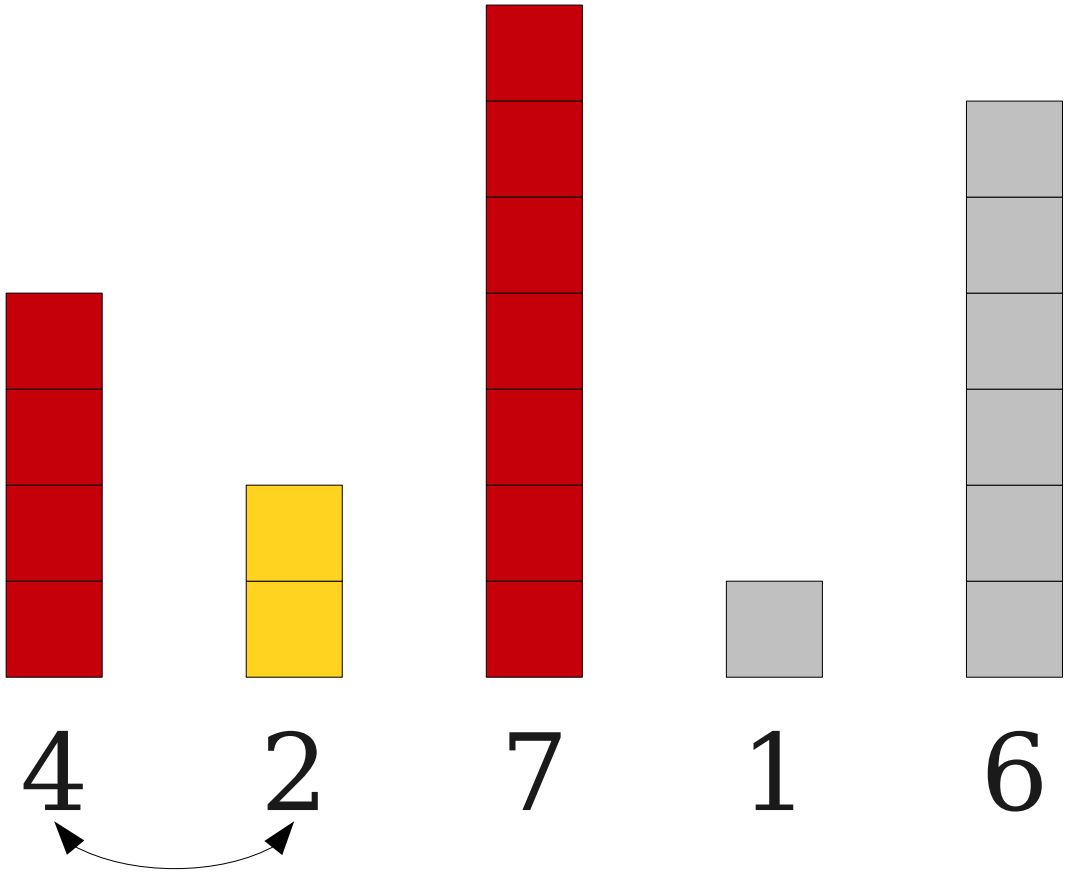
Insertion Sort



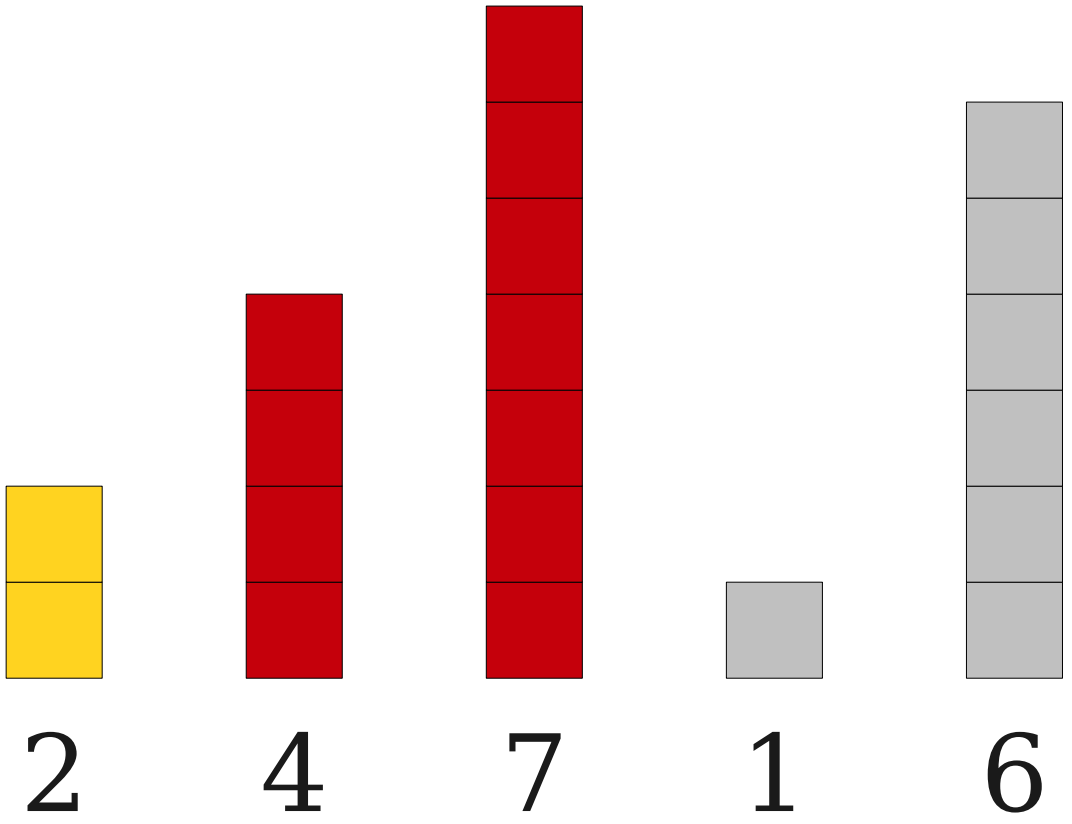
Insertion Sort



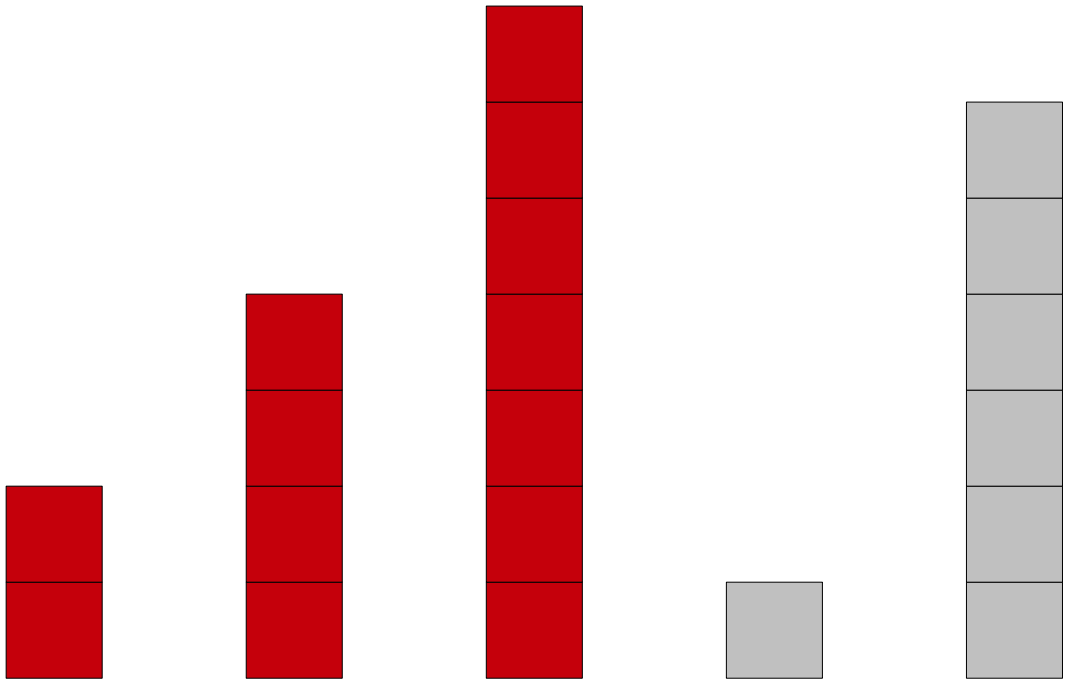
Insertion Sort



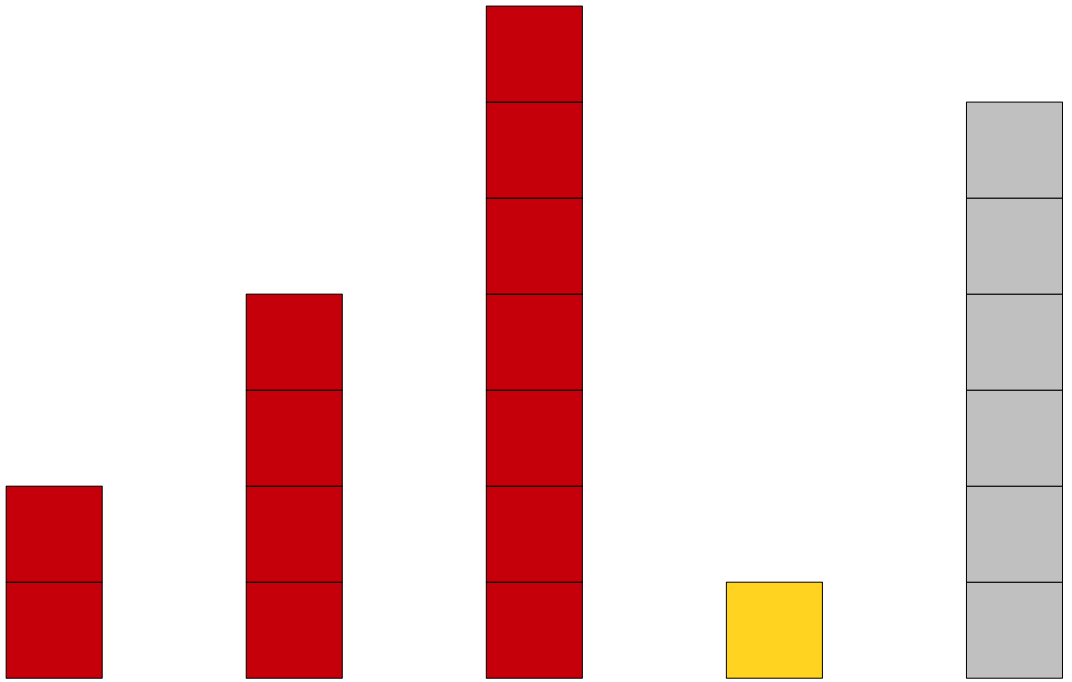
Insertion Sort



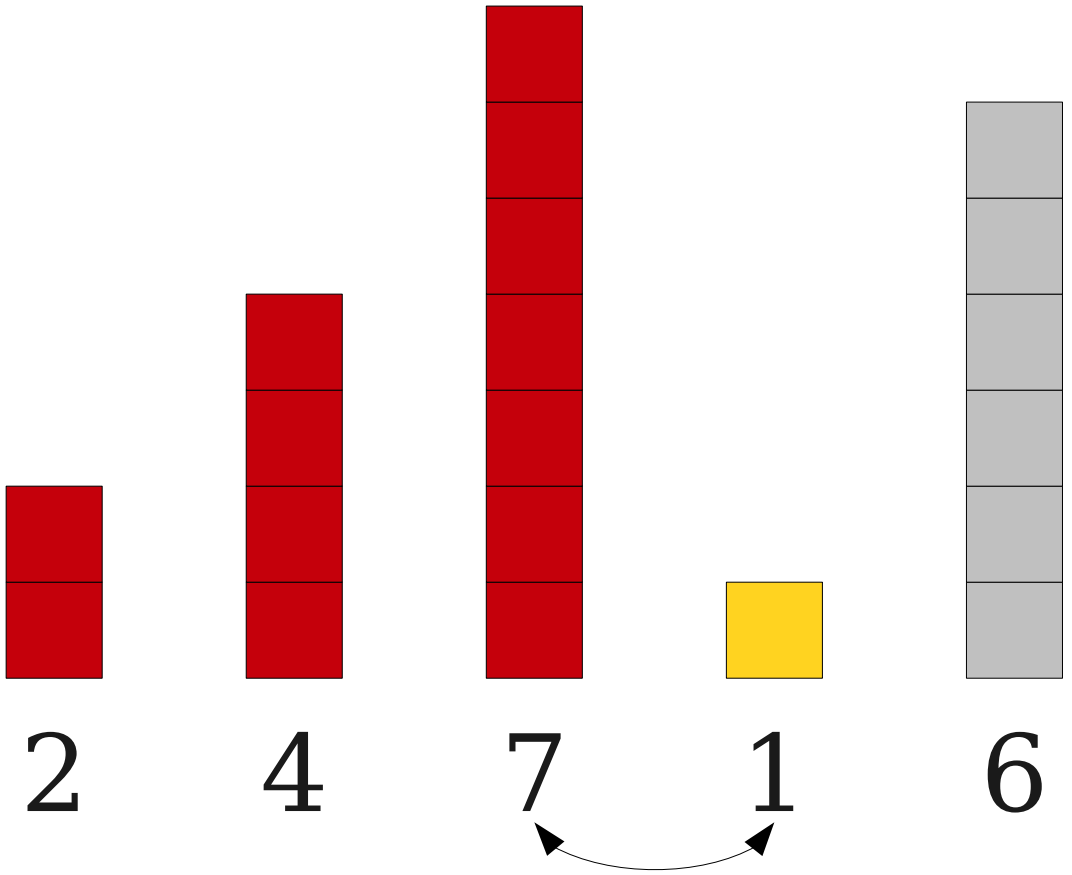
Insertion Sort



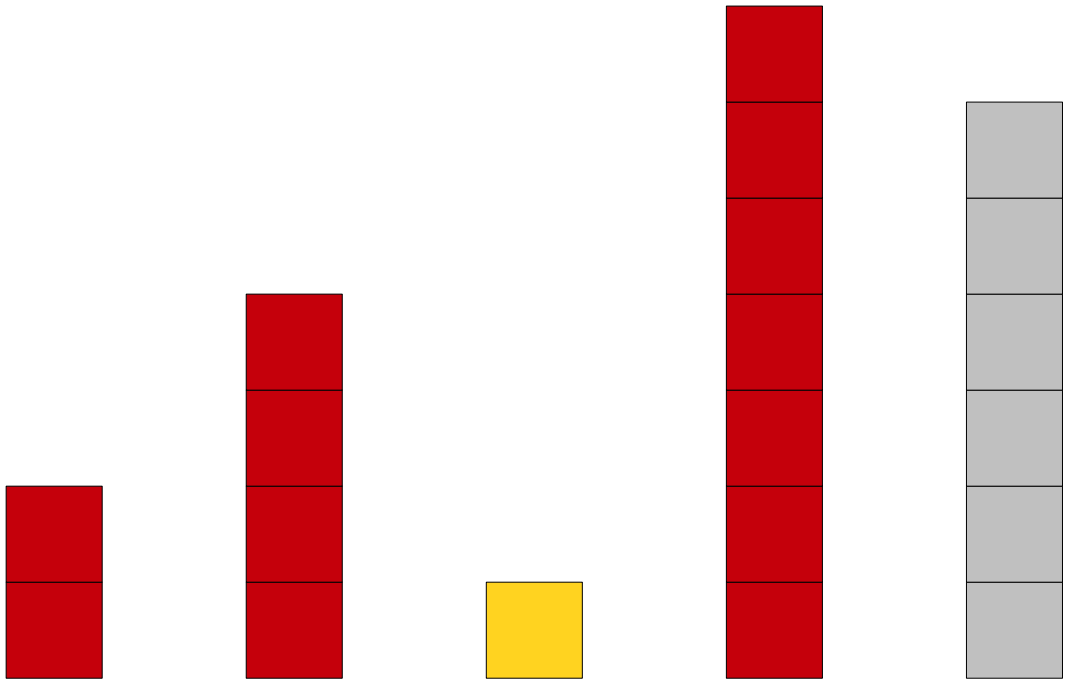
Insertion Sort



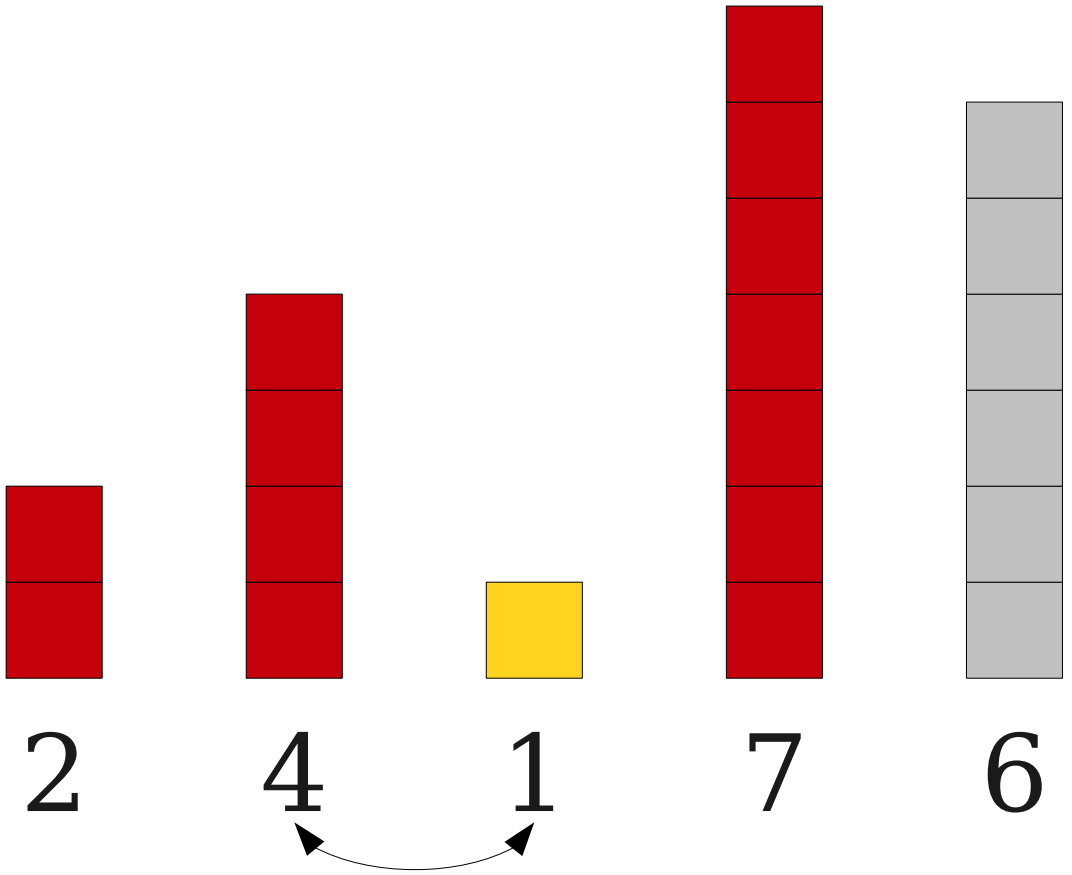
Insertion Sort



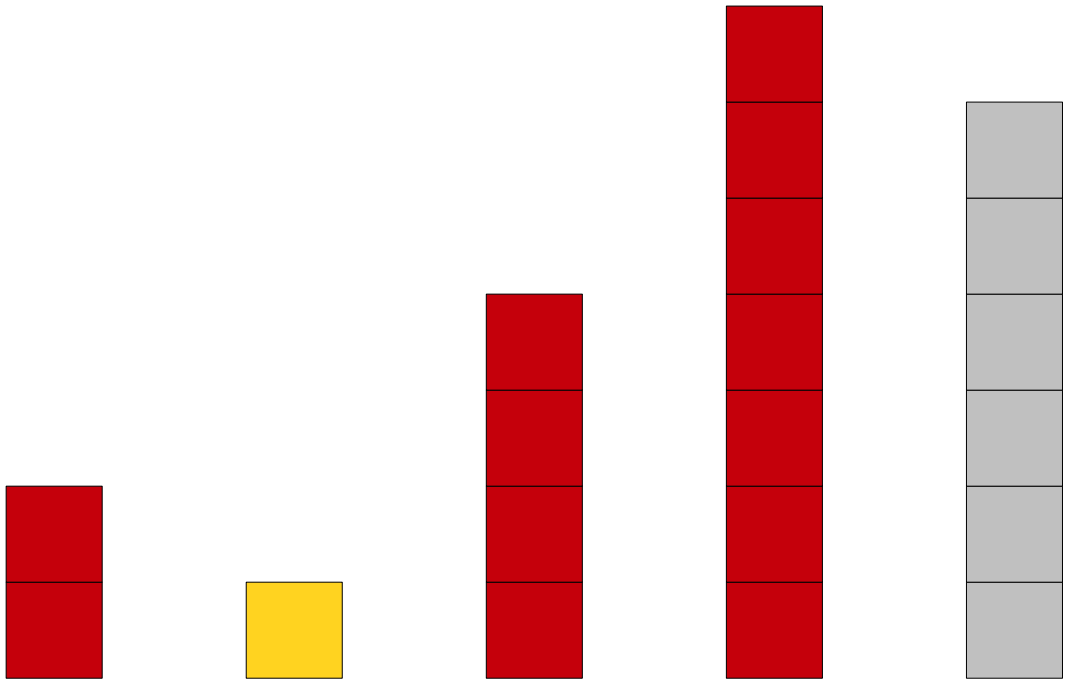
Insertion Sort



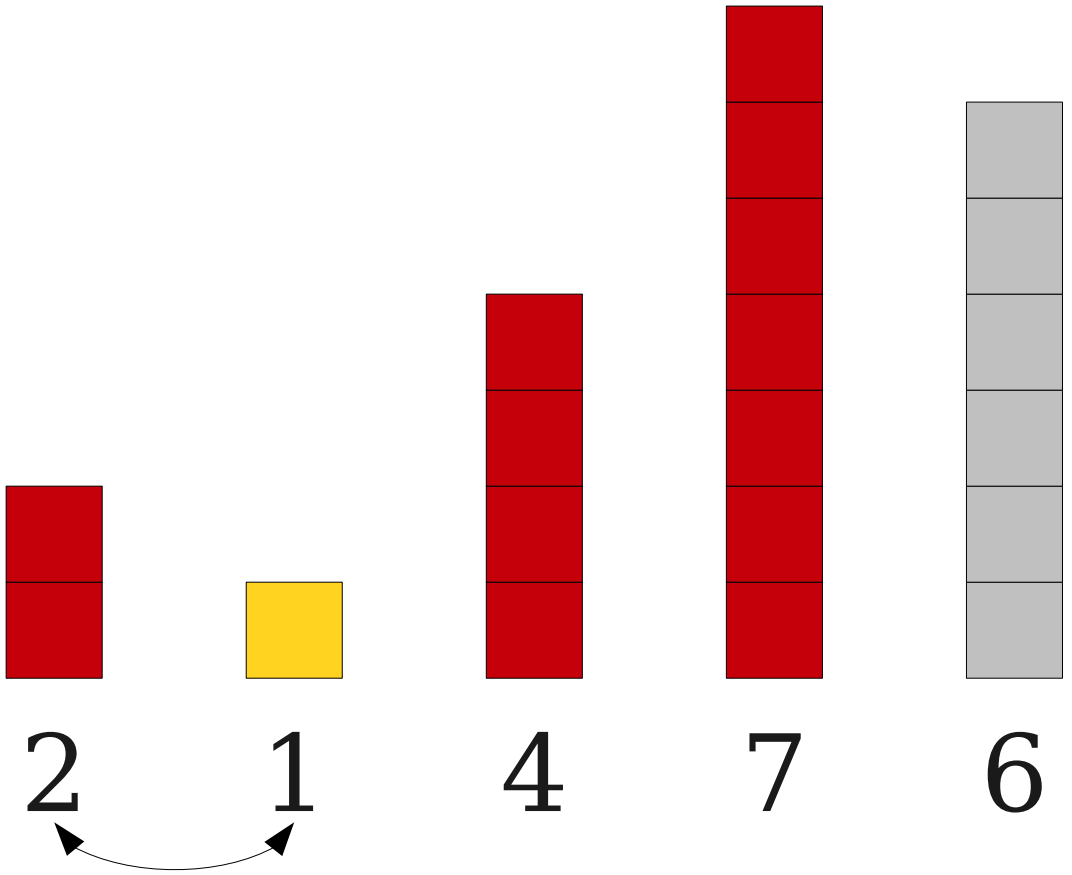
Insertion Sort



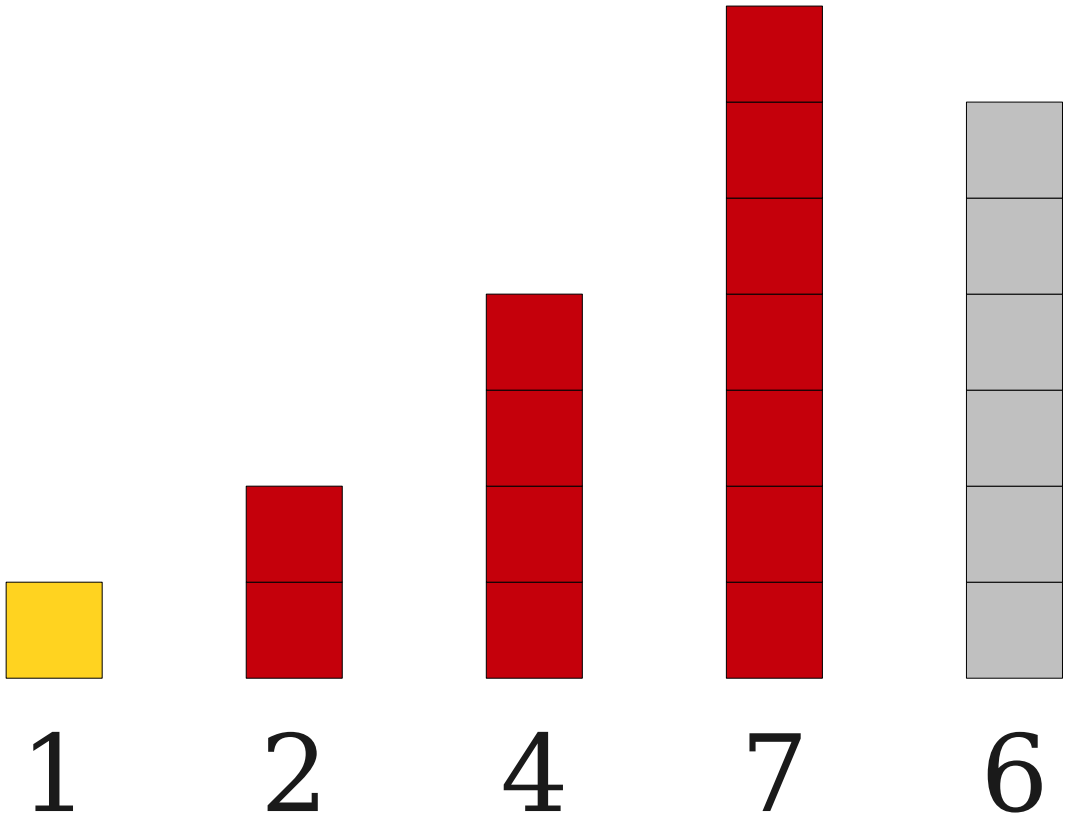
Insertion Sort



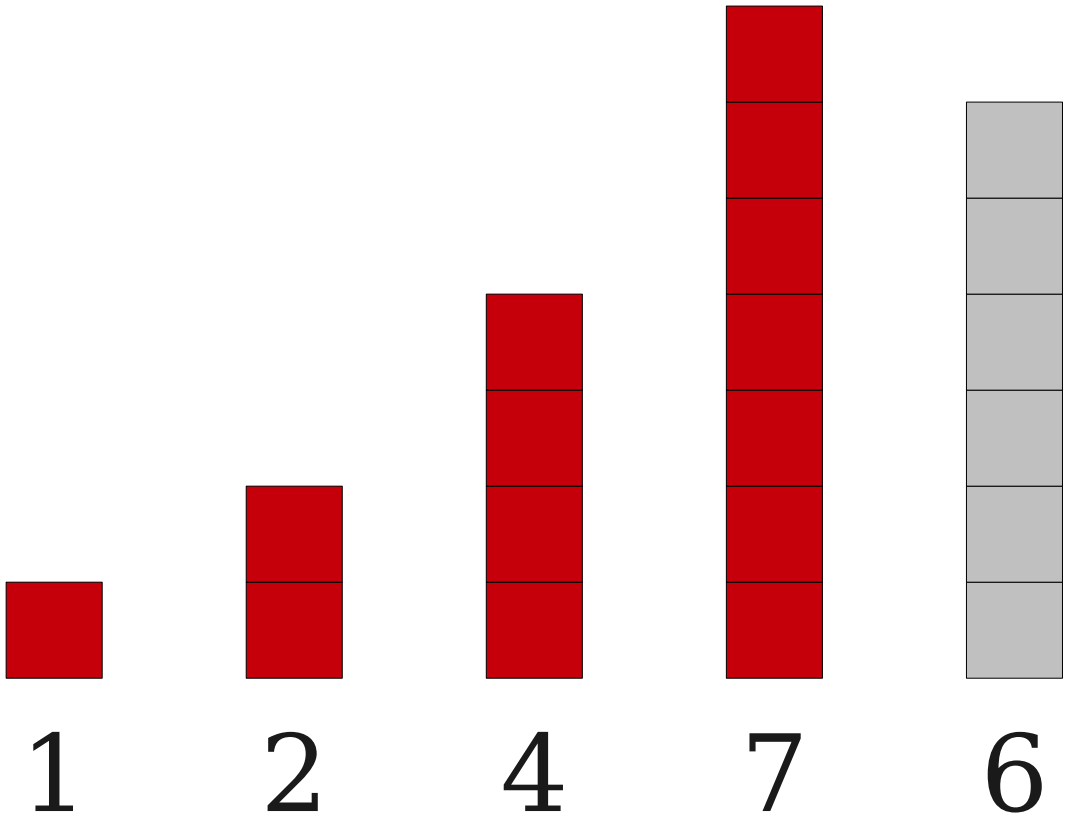
Insertion Sort



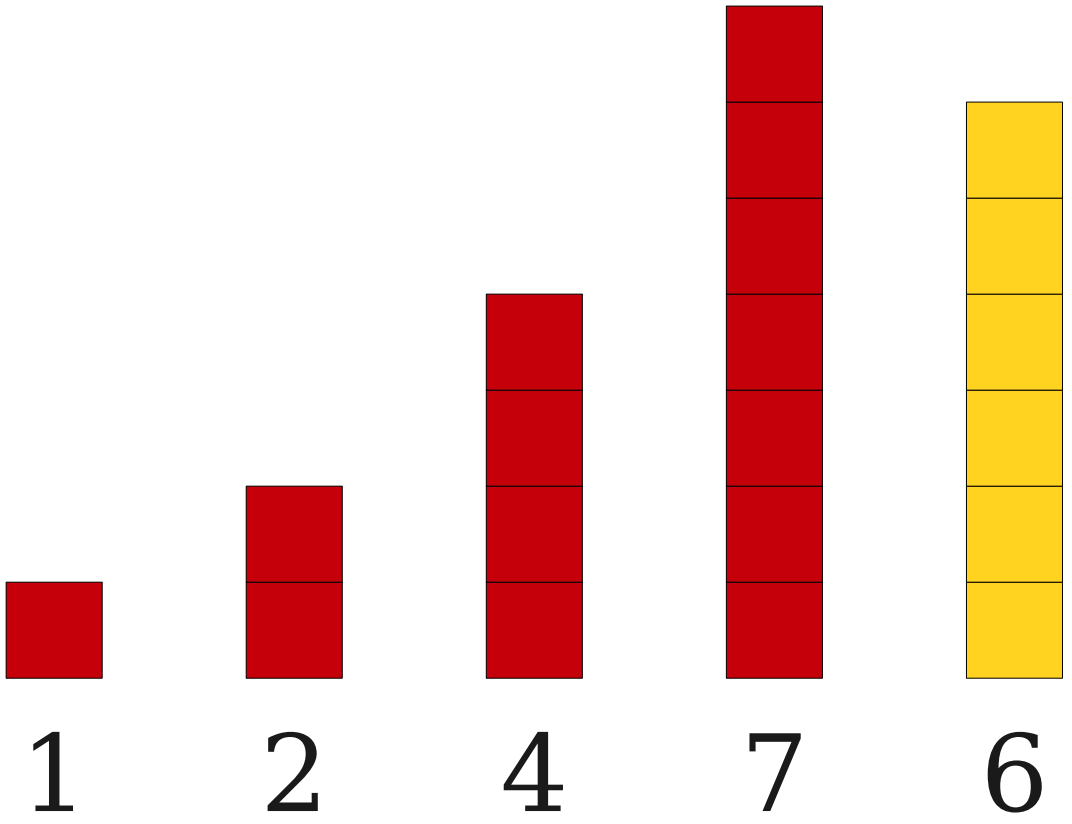
Insertion Sort



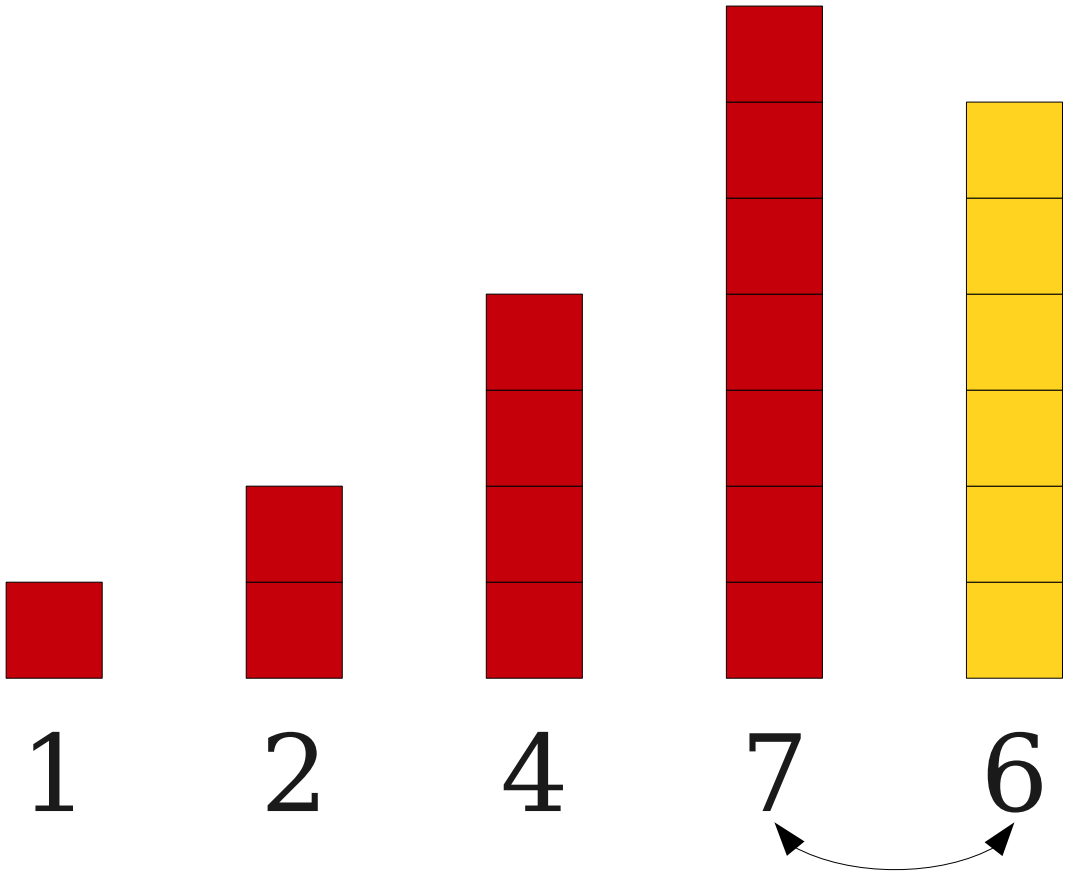
Insertion Sort



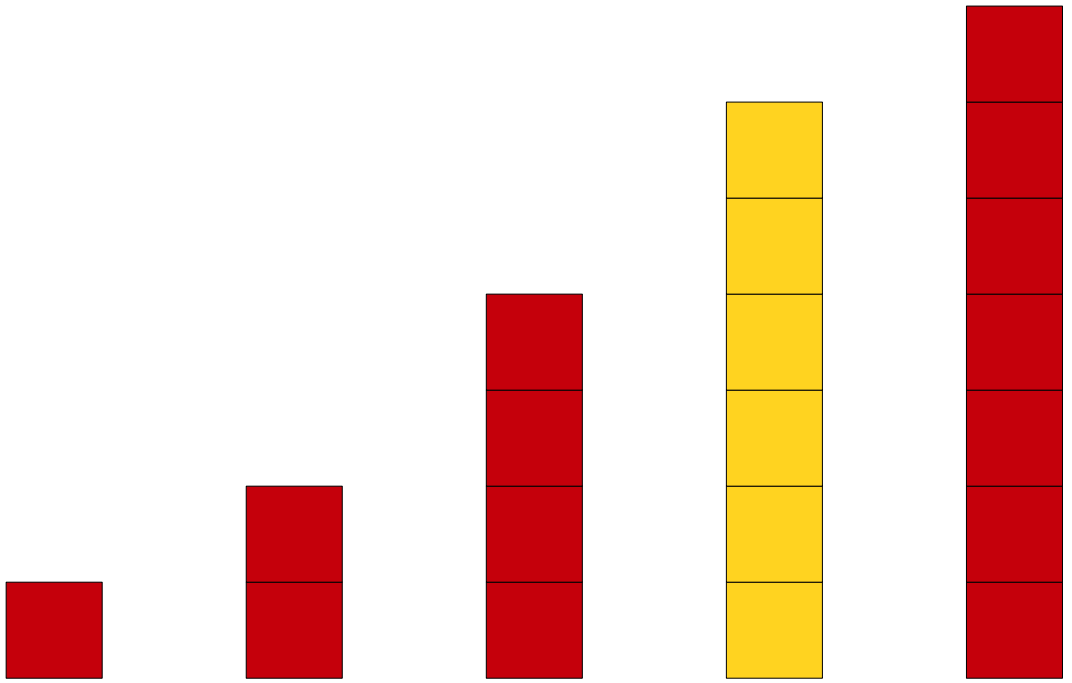
Insertion Sort



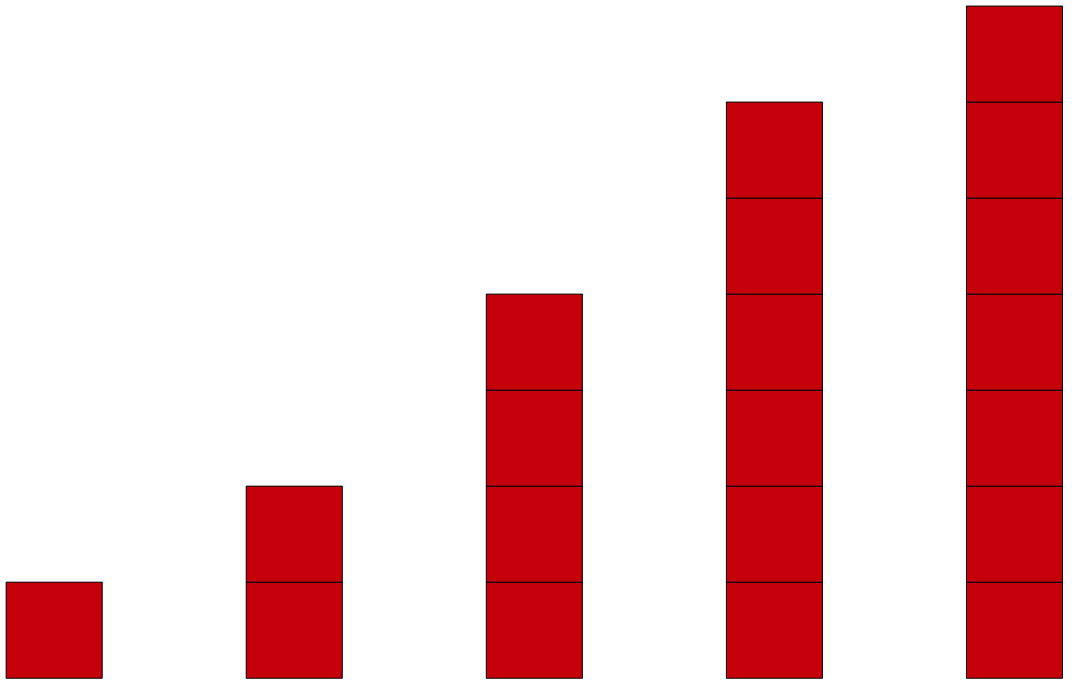
Insertion Sort



Insertion Sort



Insertion Sort



Lemma: If **`A[0 … i-1]`** is sorted at the top of the insertion sort loop, after the inner loop terminates, **`A[0 … i]`** will be sorted.

Lemma: If **`A[0 … i-1]`** is sorted at the top of the insertion sort loop, after the inner loop terminates, **`A[0 … i]`** will be sorted.

Proof: Color all values in **`A[0 … i-1]`** red and color the element at **`A[i]`** gold.

Lemma: If **`A[0 … i-1]`** is sorted at the top of the insertion sort loop, after the inner loop terminates, **`A[0 … i]`** will be sorted.

Proof: Color all values in **`A[0 … i-1]`** red and color the element at **`A[i]`** gold. Note that the following are always true:

Lemma: If **`A[0 … i-1]`** is sorted at the top of the insertion sort loop, after the inner loop terminates, **`A[0 … i]`** will be sorted.

Proof: Color all values in **`A[0 … i-1]`** red and color the element at **`A[i]`** gold. Note that the following are always true:

- The gold element is always at index **`j`**

Lemma: If **`A[0 … i-1]`** is sorted at the top of the insertion sort loop, after the inner loop terminates, **`A[0 … i]`** will be sorted.

Proof: Color all values in **`A[0 … i-1]`** red and color the element at **`A[i]`** gold. Note that the following are always true:

- The gold element is always at index **`j`**, since it starts at index **`i = j`** and each time it moves, **`j`** is updated to its new index.

Lemma: If **A[0 … i-1]** is sorted at the top of the insertion sort loop, after the inner loop terminates, **A[0 … i]** will be sorted.

Proof: Color all values in **A[0 … i-1]** red and color the element at **A[i]** gold. Note that the following are always true:

- The gold element is always at index **j**, since it starts at index **i = j** and each time it moves, **j** is updated to its new index. Since all swaps involve positions **j** and **j-1**, each swap moves the gold element.

Lemma: If **`A[0 … i-1]`** is sorted at the top of the insertion sort loop, after the inner loop terminates, **`A[0 … i]`** will be sorted.

Proof: Color all values in **`A[0 … i-1]`** red and color the element at **`A[i]`** gold. Note that the following are always true:

- The gold element is always at index **`j`**, since it starts at index **`i = j`** and each time it moves, **`j`** is updated to its new index. Since all swaps involve positions **`j`** and **`j-1`**, each swap moves the gold element.
- The red elements are always in sorted order

Lemma: If **`A[0 … i-1]`** is sorted at the top of the insertion sort loop, after the inner loop terminates, **`A[0 … i]`** will be sorted.

Proof: Color all values in **`A[0 … i-1]`** red and color the element at **`A[i]`** gold. Note that the following are always true:

- The gold element is always at index **`j`**, since it starts at index **`i = j`** and each time it moves, **`j`** is updated to its new index. Since all swaps involve positions **`j`** and **`j-1`**, each swap moves the gold element.
- The red elements are always in sorted order, since they begin sorted and every swap exchanges the gold element with an adjacent red element and thus never moves a red element across another.

Lemma: If **`A[0 … i-1]`** is sorted at the top of the insertion sort loop, after the inner loop terminates, **`A[0 … i]`** will be sorted.

Proof: Color all values in **`A[0 … i-1]`** red and color the element at **`A[i]`** gold. Note that the following are always true:

- The gold element is always at index **`j`**, since it starts at index **`i = j`** and each time it moves, **`j`** is updated to its new index. Since all swaps involve positions **`j`** and **`j-1`**, each swap moves the gold element.
- The red elements are always in sorted order, since they begin sorted and every swap exchanges the gold element with an adjacent red element and thus never moves a red element across another.
- The gold element is always less than all red elements that come after it.

Lemma: If **`A[0 … i-1]`** is sorted at the top of the insertion sort loop, after the inner loop terminates, **`A[0 … i]`** will be sorted.

Proof: Color all values in **`A[0 … i-1]`** red and color the element at **`A[i]`** gold. Note that the following are always true:

- The gold element is always at index **`j`**, since it starts at index **`i = j`** and each time it moves, **`j`** is updated to its new index. Since all swaps involve positions **`j`** and **`j-1`**, each swap moves the gold element.
- The red elements are always in sorted order, since they begin sorted and every swap exchanges the gold element with an adjacent red element and thus never moves a red element across another.
- The gold element is always less than all red elements that come after it. Initially, there are no red elements after the gold element, and each swap moves a larger red element across the gold element.

Lemma: If **`A[0 … i-1]`** is sorted at the top of the insertion sort loop, after the inner loop terminates, **`A[0 … i]`** will be sorted.

Proof: Color all values in **`A[0 … i-1]`** red and color the element at **`A[i]`** gold. Note that the following are always true:

- The gold element is always at index **`j`**, since it starts at index **`i = j`** and each time it moves, **`j`** is updated to its new index. Since all swaps involve positions **`j`** and **`j-1`**, each swap moves the gold element.
- The red elements are always in sorted order, since they begin sorted and every swap exchanges the gold element with an adjacent red element and thus never moves a red element across another.
- The gold element is always less than all red elements that come after it. Initially, there are no red elements after the gold element, and each swap moves a larger red element across the gold element.

Since the inner loop runs while **`j > 0`** and **`A[j-1] > A[j]`**, we know one of the following must be true when the loop ends:

Lemma: If **A[0 … i-1]** is sorted at the top of the insertion sort loop, after the inner loop terminates, **A[0 … i]** will be sorted.

Proof: Color all values in **A[0 … i-1]** red and color the element at **A[i]** gold. Note that the following are always true:

- The gold element is always at index **j**, since it starts at index **i = j** and each time it moves, **j** is updated to its new index. Since all swaps involve positions **j** and **j-1**, each swap moves the gold element.
- The red elements are always in sorted order, since they begin sorted and every swap exchanges the gold element with an adjacent red element and thus never moves a red element across another.
- The gold element is always less than all red elements that come after it. Initially, there are no red elements after the gold element, and each swap moves a larger red element across the gold element.

Since the inner loop runs while **j > 0** and **A[j-1] > A[j]**, we know one of the following must be true when the loop ends:

Case 1: **j = 0**.

Case 2: **A[j-1] ≤ A[j]**.

Lemma: If **A[0 … i-1]** is sorted at the top of the insertion sort loop, after the inner loop terminates, **A[0 … i]** will be sorted.

Proof: Color all values in **A[0 … i-1]** red and color the element at **A[i]** gold. Note that the following are always true:

- The gold element is always at index **j**, since it starts at index **i = j** and each time it moves, **j** is updated to its new index. Since all swaps involve positions **j** and **j-1**, each swap moves the gold element.
- The red elements are always in sorted order, since they begin sorted and every swap exchanges the gold element with an adjacent red element and thus never moves a red element across another.
- The gold element is always less than all red elements that come after it. Initially, there are no red elements after the gold element, and each swap moves a larger red element across the gold element.

Since the inner loop runs while **j > 0** and **A[j-1] > A[j]**, we know one of the following must be true when the loop ends:

Case 1: **j = 0**. Since the gold element is less than all elements after it (which are red) and the red elements are sorted, **A[0 … i]** is sorted.

Case 2: **A[j-1] ≤ A[j]**.

Lemma: If **A[0 … i-1]** is sorted at the top of the insertion sort loop, after the inner loop terminates, **A[0 … i]** will be sorted.

Proof: Color all values in **A[0 … i-1]** red and color the element at **A[i]** gold. Note that the following are always true:

- The gold element is always at index **j**, since it starts at index **i = j** and each time it moves, **j** is updated to its new index. Since all swaps involve positions **j** and **j-1**, each swap moves the gold element.
- The red elements are always in sorted order, since they begin sorted and every swap exchanges the gold element with an adjacent red element and thus never moves a red element across another.
- The gold element is always less than all red elements that come after it. Initially, there are no red elements after the gold element, and each swap moves a larger red element across the gold element.

Since the inner loop runs while **j > 0** and **A[j-1] > A[j]**, we know one of the following must be true when the loop ends:

Case 1: **j = 0**. Since the gold element is less than all elements after it (which are red) and the red elements are sorted, **A[0 … i]** is sorted.

Case 2: **A[j-1] ≤ A[j]**. Since the red elements are sorted and the gold element is at least as large as the red element that precedes it, it is no smaller than any preceding red elements. It is also smaller than all successive red elements. Thus **A[0 … i]** is sorted.

Lemma: If **A[0 … i-1]** is sorted at the top of the insertion sort loop, after the inner loop terminates, **A[0 … i]** will be sorted.

Proof: Color all values in **A[0 … i-1]** red and color the element at **A[i]** gold. Note that the following are always true:

- The gold element is always at index **j**, since it starts at index **i = j** and each time it moves, **j** is updated to its new index. Since all swaps involve positions **j** and **j-1**, each swap moves the gold element.
- The red elements are always in sorted order, since they begin sorted and every swap exchanges the gold element with an adjacent red element and thus never moves a red element across another.
- The gold element is always less than all red elements that come after it. Initially, there are no red elements after the gold element, and each swap moves a larger red element across the gold element.

Since the inner loop runs while **j > 0** and **A[j-1] > A[j]**, we know one of the following must be true when the loop ends:

Case 1: **j = 0**. Since the gold element is less than all elements after it (which are red) and the red elements are sorted, **A[0 … i]** is sorted.

Case 2: **A[j-1] ≤ A[j]**. Since the red elements are sorted and the gold element is at least as large as the red element that precedes it, it is no smaller than any preceding red elements. It is also smaller than all successive red elements. Thus **A[0 … i]** is sorted. ■

Proof Structure

- At a high level, the proof works by finding three invariants (the position of the gold element, the relative ordering of the red elements, and the relation of the gold element to the successive red elements).
- The proof works by establishing these are indeed invariants (through informal inductive arguments), then showing that these invariants show the overall result.
- Note the level of detail – this proof could be made significantly more rigorous and complex, but sufficiently justifies its main points and show how they lead to an overall result.
- It takes practice to write proofs like this, and we'll try to give you feedback in the problem sets to get you to write clean proofs.

Theorem: Insertion sort sorts its argument.

Proof: We will prove that after any number of iterations of the top-level loop, **A[0 … i-1]** is sorted. Since the loop terminates with **i = length(A)**, this implies **A[0 … length(A)-1]** is sorted when the loop ends, proving the theorem.

To see this, note that before the loop starts, when **i = 0**, **A[0 … i-1] = A[0 … -1]** is trivially sorted. Otherwise, let $\mathbf{i_0}$ be the value of **i** at the start of some loop iteration. If **A[0 … $\mathbf{i_0}$-1]** is sorted at the start of that loop iteration, then by our lemma **A[0 … $\mathbf{i_0}$]** will be sorted at the end of the iteration. Since **i = $\mathbf{i_0}$+1** at the end of the loop, this means **A[0 … i-1]** is still sorted at the end of the loop. ■

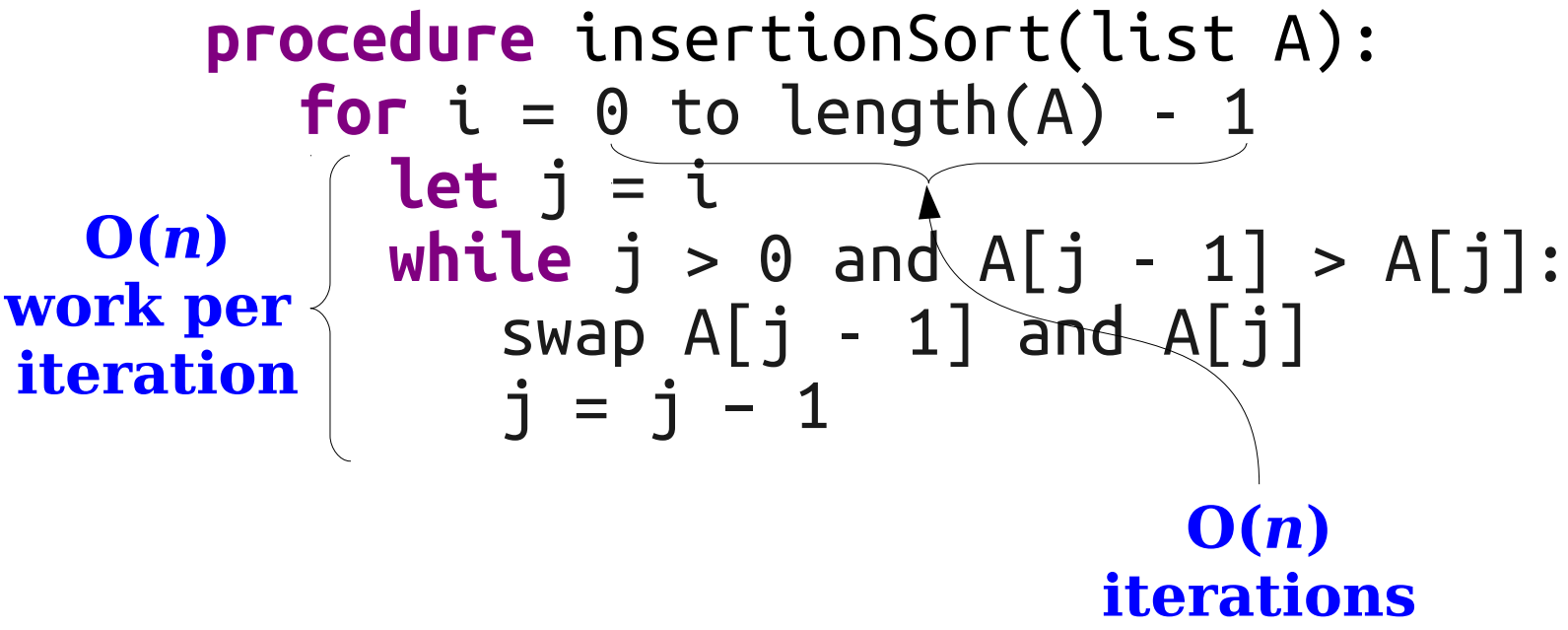
Question 1: How do we prove this always sorts the input array?

Question 2: How *efficiently* does insertion sort sort the input array?

Question 1: How do we prove this always sorts the input array?

Question 2: How *efficiently* does insertion sort sort the input array?

Analyzing the Runtime



Total work done:

$O(n^2)$

What does big-O notation mean?

Big-O Notation

- **Big-O notation** is a mathematical notation for upper-bounding a function's growth rate.
- Informally, can be found by ignoring constants and non-dominant growth terms.
- Examples:
  - $n + 137 = \mathbf{O(n)}$
  - $3n + 42 = \mathbf{O(n)}$
  - $n^2 + 3n - 2 = \mathbf{O(n^2)}$
  - $n^3 + 10n^2 \log n - 15n = \mathbf{O(n^3)}$
  - $2^n + n^2 = \mathbf{O(2^n)}$

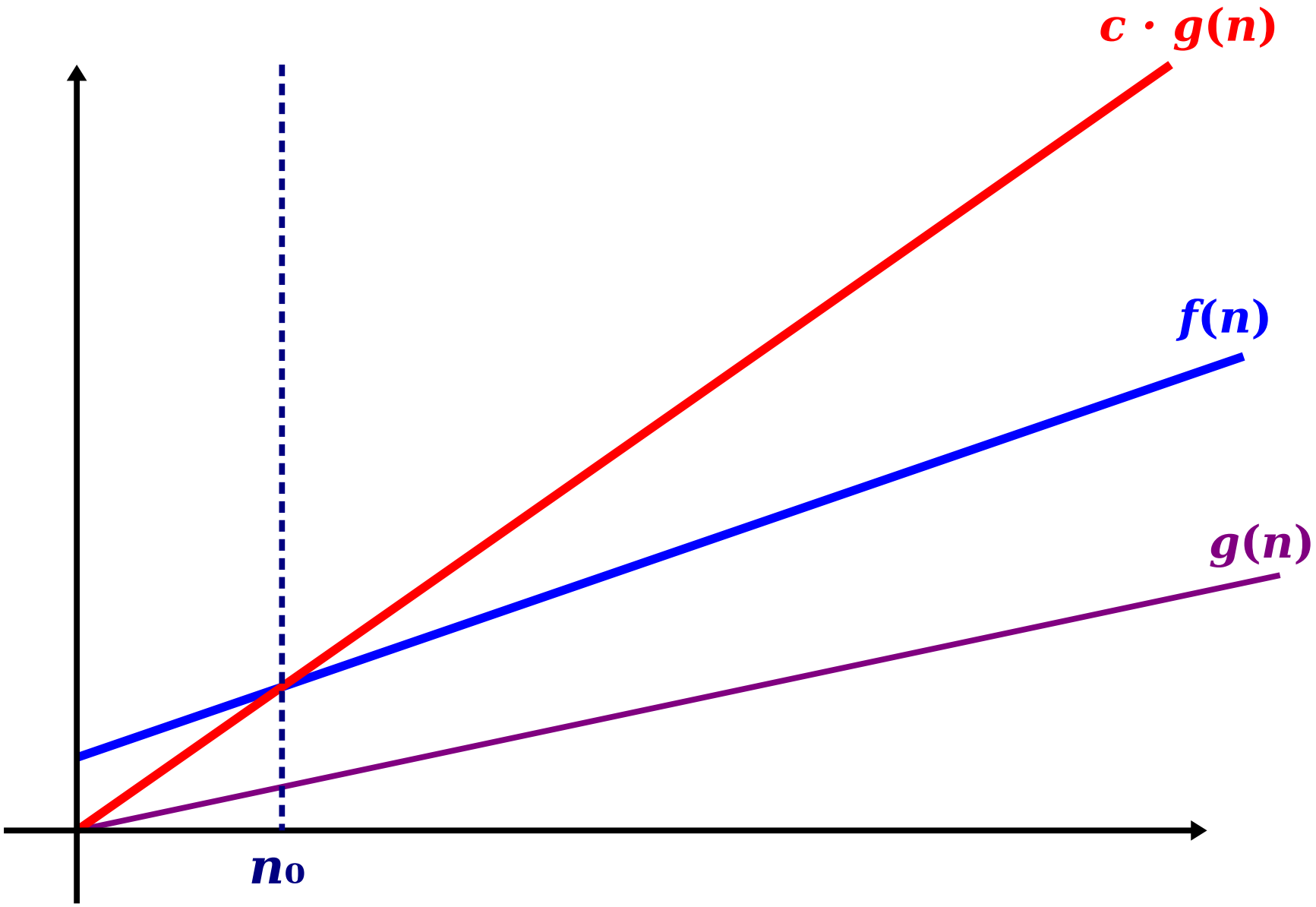
Big-O Notation, Formally

- Formally speaking, let $f, g : \mathbb{N} \to \mathbb{N}$.
- We say $\boldsymbol{f(n) = O(g(n))}$ iff

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}.$$
$$\forall n \in \mathbb{N}.$$
$$(n \geq n_0 \to f(n) \leq c \cdot g(n))$$

- Intuitively, when n gets "sufficiently large" (i.e. greater than n_0), $f(n)$ is bounded from above by some constant multiple (specifically, c) of $g(n)$.

$$f(n) = O(g(n))$$



Big-O and Runtimes

- This definition of big-O notation talks about *functions*, not *runtimes*.
- How do we connect them together?
- For any algorithm A that works on inputs of size n, there is some function $w_A(n)$ that gives the maximum amount of work A can do on an input of size n.
  - We might not know what this function is, but mathematically it's guaranteed to exist.
- When we say an algorithm A has worst-case runtime $O(n^2)$, for example, we're saying that $w_A(n) = O(n^2)$.

Big-O Notation, Formally

- Insertion sort never executes any more than $3n^2 + 2n + 1$ lines of code.

  Claim: $3n^2 + 2n + 1 = O(n^2)$.

Big-O Notation, Formally

- Insertion sort never executes any more than $3n^2 + 2n + 1$ lines of code.

  Claim: $3n^2 + 2n + 1 = O(n^2)$.

  Proof: Take $n_0 = 1$ and $c = 6$.

Big-O Notation, Formally

- Insertion sort never executes any more than $3n^2 + 2n + 1$ lines of code.

  Claim: $3n^2 + 2n + 1 = O(n^2)$.

  Proof: Take $n_0 = 1$ and $c = 6$. Then for any $n \geq n_0$, we have

  $3n^2 + 2n + 1$

Big-O Notation, Formally

- Insertion sort never executes any more than $3n^2 + 2n + 1$ lines of code.

  Claim: $3n^2 + 2n + 1 = O(n^2)$.

  Proof: Take $n_0 = 1$ and $c = 6$. Then for any $n \geq n_0$, we have

$$3n^2 + 2n + 1 \leq 3n^2 + 2n \cdot n + 1 \cdot n^2$$

Big-O Notation, Formally

- Insertion sort never executes any more than $3n^2 + 2n + 1$ lines of code.

  Claim: $3n^2 + 2n + 1 = O(n^2)$.

  Proof: Take $n_0 = 1$ and $c = 6$. Then for any $n \geq n_0$, we have

$$\begin{aligned} 3n^2 + 2n + 1 &\leq 3n^2 + 2n \cdot n + 1 \cdot n^2 \\ &= 3n^2 + 2n^2 + n^2 \end{aligned}$$

Big-O Notation, Formally

- Insertion sort never executes any more than $3n^2 + 2n + 1$ lines of code.

  Claim: $3n^2 + 2n + 1 = O(n^2)$.

  Proof: Take $n_0 = 1$ and $c = 6$. Then for any $n \geq n_0$, we have

$$\begin{aligned} 3n^2 + 2n + 1 &\leq 3n^2 + 2n \cdot n + 1 \cdot n^2 \\ &= 3n^2 + 2n^2 + n^2 \\ &= 6n^2 \end{aligned}$$

Big-O Notation, Formally

- Insertion sort never executes any more than $3n^2 + 2n + 1$ lines of code.

  Claim: $3n^2 + 2n + 1 = O(n^2)$.

  Proof: Take $n_0 = 1$ and $c = 6$. Then for any $n \geq n_0$, we have

$$\begin{aligned} 3n^2 + 2n + 1 &\leq 3n^2 + 2n \cdot n + 1 \cdot n^2 \\ &= 3n^2 + 2n^2 + n^2 \\ &= 6n^2 \\ &\leq 6(n^2) \end{aligned}$$

Big-O Notation, Formally

- Insertion sort never executes any more than $3n^2 + 2n + 1$ lines of code.

  Claim: $3n^2 + 2n + 1 = O(n^2)$.

  Proof: Take $n_0 = 1$ and $c = 6$. Then for any $n \geq n_0$, we have

$$\begin{aligned} 3n^2 + 2n + 1 &\leq 3n^2 + 2n \cdot n + 1 \cdot n^2 \\ &= 3n^2 + 2n^2 + n^2 \\ &= 6n^2 \\ &\leq 6(n^2) \ \blacksquare \end{aligned}$$

Big-O Notation, Formally

- Insertion sort never executes any more than $3n^2 + 2n + 1$ lines of code.

  Claim: $3n^2 + 2n + 1 = O(n^{161})$.

  Proof: Take $n_0 = 1$ and $c = 6$. Then for any $n \geq n_0$, we have

$$\begin{aligned} 3n^2 + 2n + 1 &\leq 3n^2 \cdot n^{159} + 2n \cdot n^{160} + 1 \cdot n^{161} \\ &= 3n^{161} + 2n^{161} + n^{161} \\ &= 6n^{161} \\ &= 6(n^{161}) \ \blacksquare \end{aligned}$$

Big-O does not guarantee a tight bound!

A Different Analysis

- What is the *best-case* runtime for insertion sort?

```
procedure insertionSort(list A):
  for i = 0 to length(A) - 1
     let j = i
     while j > 0 and A[j - 1] > A[j]:
        swap A[j - 1] and A[j]
        j = j - 1
```

A Different Analysis

- What is the *best-case* runtime for insertion sort?

```
procedure insertionSort(list A):
  for i = 0 to length(A) - 1
     let j = i
     while j > 0 and A[j - 1] > A[j]:
        swap A[j - 1] and A[j]
        j = j - 1
```

- What happens if the input is already sorted?

A Different Analysis

- What is the *best-case* runtime for insertion sort?

```
procedure insertionSort(list A):
  for i = 0 to length(A) - 1
     let j = i
     while j > 0 and A[j - 1] > A[j]:
        swap A[j - 1] and A[j]
        j = j - 1
```

- What happens if the input is already sorted?
- (For comparison: what happens if the input is reverse-sorted?)

A Different Analysis

- True but not very precise statement:

In the best case,
insertion sort does O(n) work.

- Why is this imprecise?
- Big-O gives an upper bound!

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}.$$
$$\forall n \in \mathbb{N}.$$
$$(n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$$

- Saying the best-case runtime is O(n) doesn't preclude a best-case runtime of 1 or log n, for example.

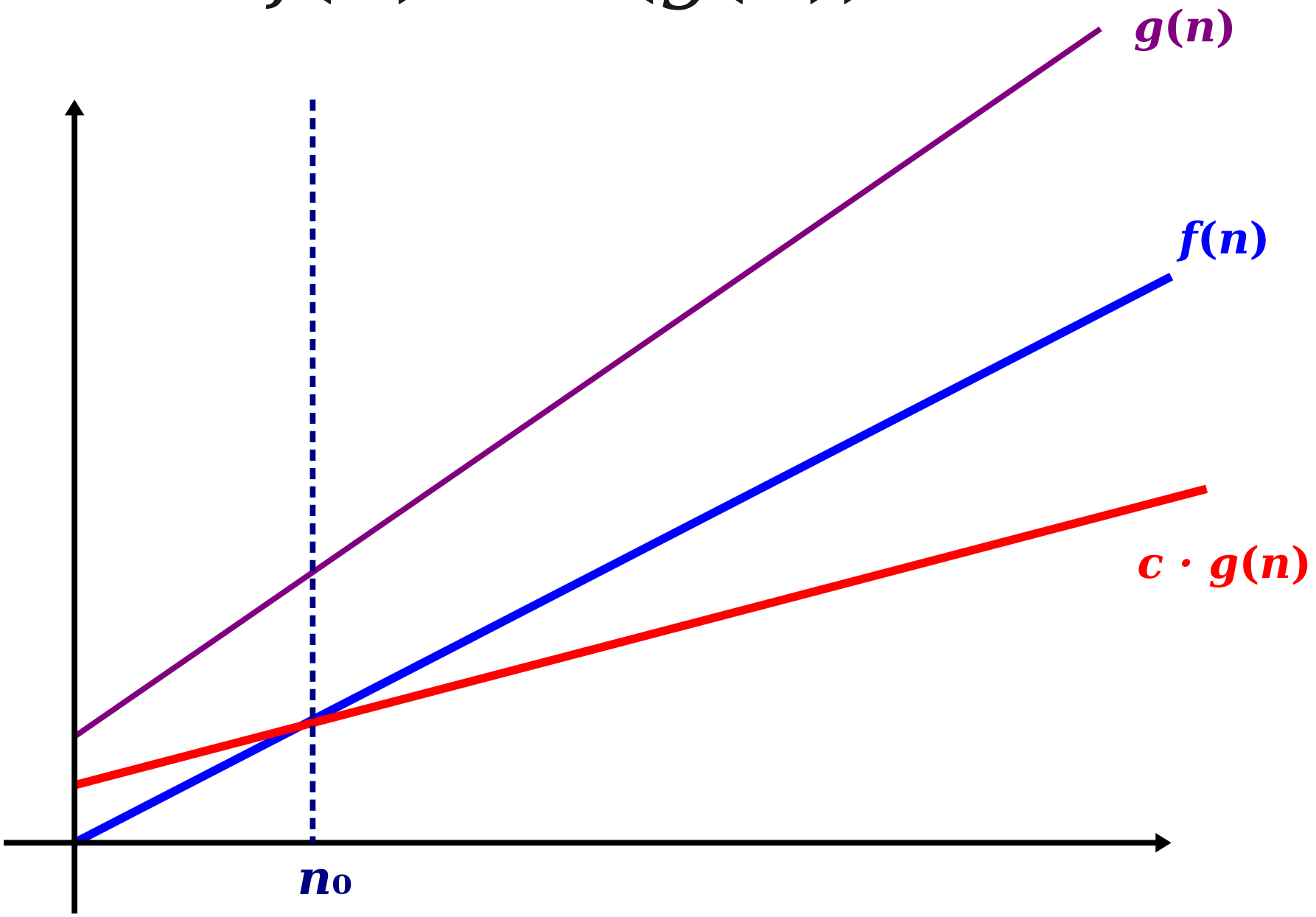
Ω Notation

- Let $f, g : \mathbb{N} \to \mathbb{N}$.
- We say $\boldsymbol{f(n) = \Omega(g(n))}$ iff

$$\boldsymbol{\exists n_0 \in \mathbb{N}, c > 0 \in \mathbb{R}.}$$
$$\boldsymbol{\forall n \in \mathbb{N}.}$$
$$\boldsymbol{(n \geq n_0 \to f(n) \geq c \cdot g(n))}$$

- Intuitively, when n gets "sufficiently large" (i.e. greater than n_0), $f(n)$ is bounded **from below** by some constant multiple (specifically, c) of $g(n)$.
- In our case, insertion sort runs in time $\boldsymbol{\Omega(n)}$.

$$f(n) = \Omega(g(n))$$



Θ Notation

- We say ***f(n) = Θ(g(n))*** iff both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- Intuitively, $f(n)$ and $g(n)$ grow at the same rate as one another.
- Examples:
  - $10n + 137 = \Theta(n)$
  - $n = \Theta(10n + 137)$
  - $15 \log (n + 3) = \Theta(\log n)$

The Story So Far

- A quick glance at the pseudocode tells us that

  Insertion sort runs in time $O(n^2)$

  Insertion sort runs in time $\Omega(n)$

- By finding concrete examples of best-case and worst-case inputs to insertion sort, we now know

  The worst-case runtime of insertion sort is $\Theta(n^2)$

  The best-case runtime of insertion sort is $\Theta(n)$

- What about the *average-case?*

Defining our Inputs

- To perform an average-case analysis, we need to define some distribution over our inputs.
- Insertion sort doesn't care about the *absolute* values of the array elements; just their *relative* values.
- Assume the input to be sorted is a random permutation of 1 … n.
  - This doesn't account for duplicate values; for now, we'll ignore that.
- In this case, how fast will insertion sort be?

```
procedure insertionSort(list A):
   for i = 0 to length(A) - 1
      let j = i
      while j > 0 and A[j - 1] > A[j]:
         swap A[j - 1] and A[j]
         j = j - 1
```

```
 procedure insertionSort(list A):
1.  for i = 0 to length(A) - 1
2.     let j = i
3.     while j > 0 and A[j - 1] > A[j]:
4.        swap A[j - 1] and A[j]
5.        j = j - 1
```

```
 procedure insertionSort(list A):
1.  for i = 0 to length(A) - 1
2.     let j = i
3.     while j > 0 and A[j - 1] > A[j]:
4.        swap A[j - 1] and A[j]
5.        j = j - 1
```

1 2 3 1 2 3 4 5 3 1 2 3 4 5 3 4 5 3 1

```
 procedure insertionSort(list A):
1.  for i = 0 to length(A) - 1
2.     let j = i
3.     while j > 0 and A[j - 1] > A[j]:
4.        swap A[j - 1] and A[j]
5.        j = j - 1
```

1 2 3 **1 2** 3 4 5 3 **1 2** 3 4 5 3 4 5 3 **1**

```
 procedure insertionSort(list A):
1.  for i = 0 to length(A) - 1
2.     let j = i
3.     while j > 0 and A[j - 1] > A[j]:
4.        swap A[j - 1] and A[j]
5.        j = j - 1
```

1 2 **3** 1 2 **3 4 5** 3 1 2 **3 4 5** 3 **4 5** 3 1

A More Precise Analysis

- Our original analysis had this form:
  - Each time the outer loop executes, the inner loop does O(n) work.
  - The outer loop runs O(n) times.
  - Therefore, the total work done is O(n^2).
- We can be much more precise in our analysis by noting the following:
  - *Across the entire run of the algorithm*, the code purely in the outer loop runs some number of times.
  - *Across the entire run of the algorithm*, the code purely inside the inner loop runs some number of times.
  - The total work *across the entire run of the algorithm* is the sum of these two amounts.

```
procedure insertionSort(list A):
   for i = 0 to length(A) - 1
      let j = i
      while j > 0 and A[j - 1] > A[j]:
         swap A[j - 1] and A[j]
         j = j - 1
```

```
procedure insertionSort(list A):
   for i = 0 to length(A) - 1
      let j = i
      while j > 0 and A[j - 1] > A[j]:
         swap A[j - 1] and A[j]
         j = j - 1
```

```
procedure insertionSort(list A):
   for i = 0 to length(A) - 1
      let j = i
      while j > 0 and A[j - 1] > A[j]:
         swap A[j - 1] and A[j]
         j = j - 1
```

Work done across the entire algorithm:

$\Theta(n)$

```
procedure insertionSort(list A):
   for i = 0 to length(A) - 1
      let j = i
      while j > 0 and A[j - 1] > A[j]:
         swap A[j - 1] and A[j]
         j = j - 1
```

```
procedure insertionSort(list A):
   for i = 0 to length(A) - 1
      let j = i
      while j > 0 and A[j - 1] > A[j]:
         swap A[j - 1] and A[j]
         j = j - 1
```

Work done across the entire algorithm:

$\Theta(n + \#swaps)$

Insertion Sort Runtime

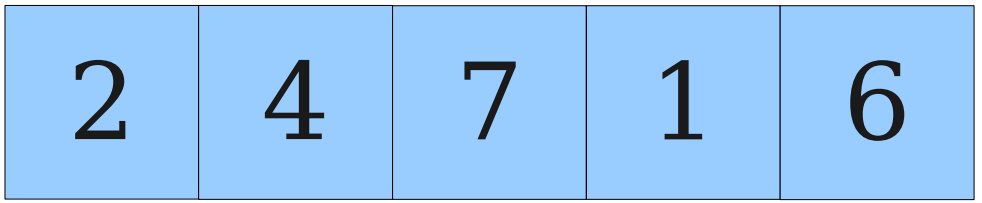
- The runtime of insertion sort is

  $$\Theta(n) + \Theta(n + \#\text{swaps}) = \mathbf{\Theta(n + \#swaps)}$$

- Matches our intuition:
  - In a sorted array, no swaps need to be made. The runtime is $\Theta(n)$.
  - In a reverse-sorted array, $\Theta(n^2)$ swaps need to be made. The runtime is $\Theta(n^2)$.
- The number of swaps is somehow connected to the "sortedness" of the original array.
- How can we measure this?

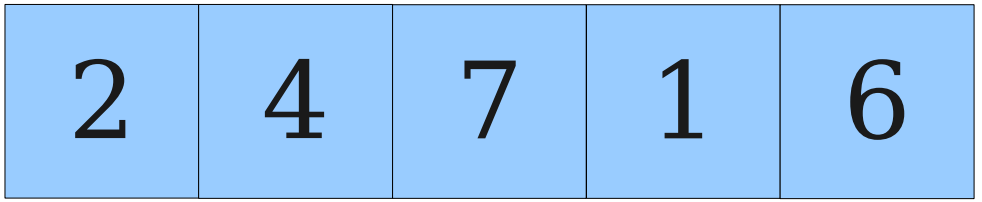
Inversions

- An **inversion** in an array `A` is a pair of elements `(A[i], A[j])` such that `i < j` (the first element appears before the second), but `A[i] > A[j]` (the elements are out of order).
- How many inversions are there in this array?



Inversions

- An **inversion** in an array `A` is a pair of elements `(A[i], A[j])` such that `i < j` (the first element appears before the second), but `A[i] > A[j]` (the elements are out of order).
- How many inversions are there in this array?

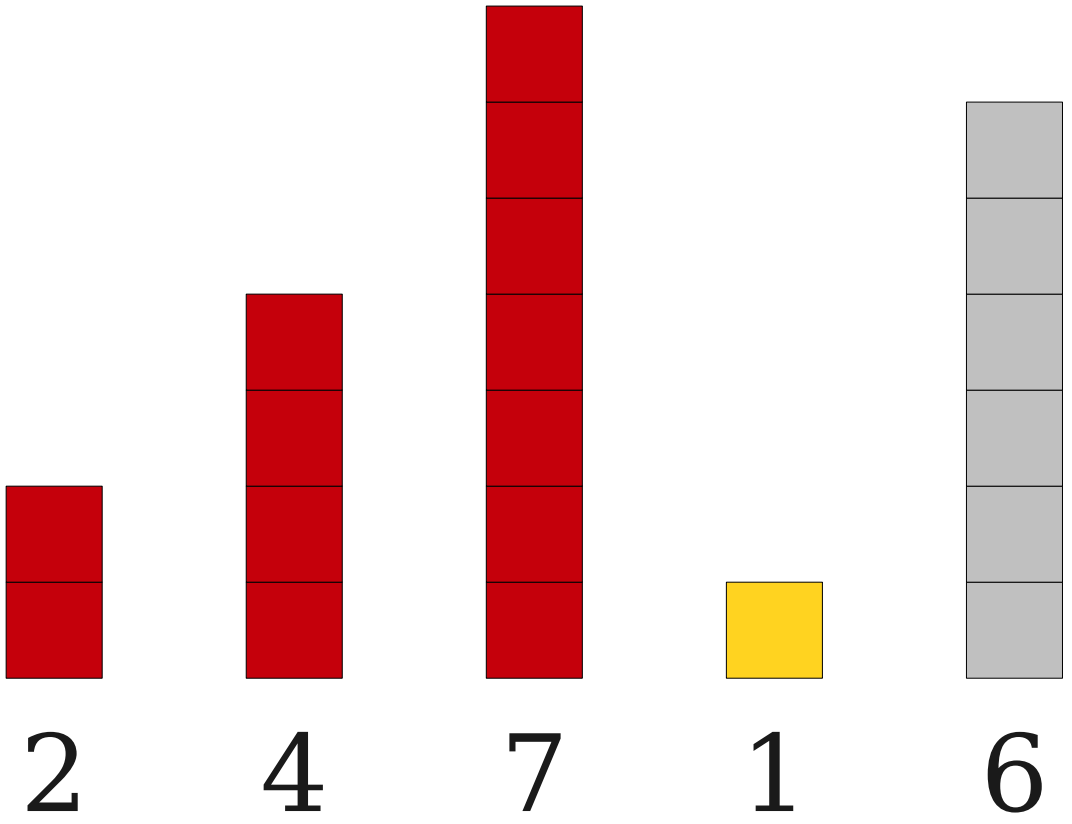


- How many inversions are there in a sorted array?

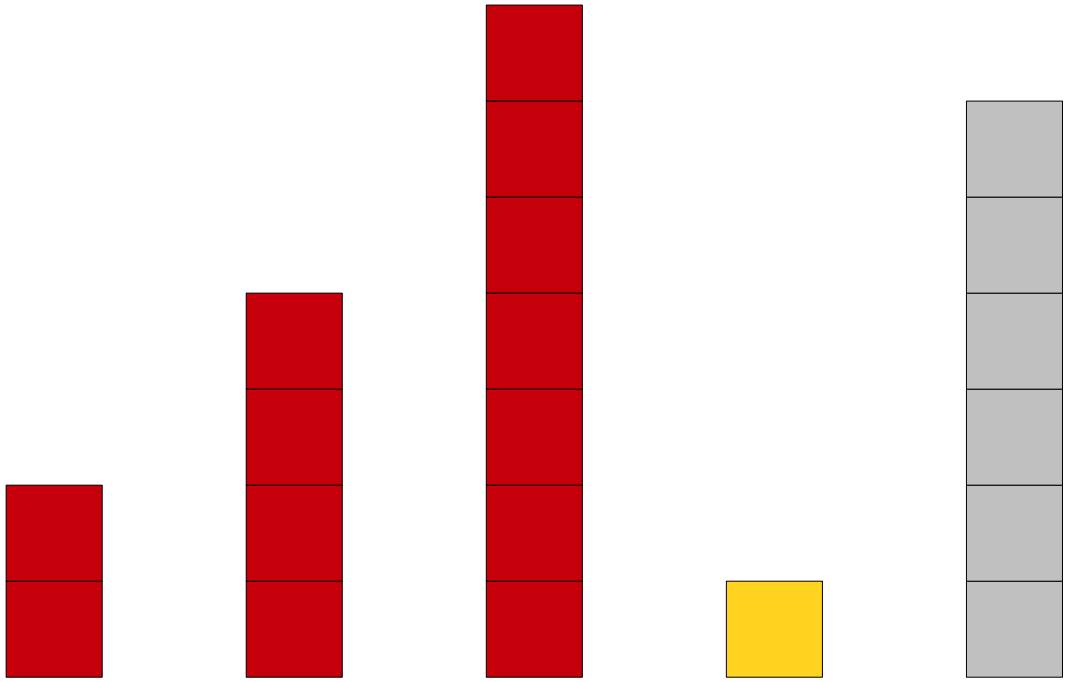
Inversions and Insertion Sort

- **Key Observation:** An unsorted array contains some number of inversions, while a sorted array contains 0.
- Every sorting algorithm ultimately decreases the number of inversions in the array to zero.
- How *efficiently* do these algorithms decrease the number of inversions?

Counting Inversions

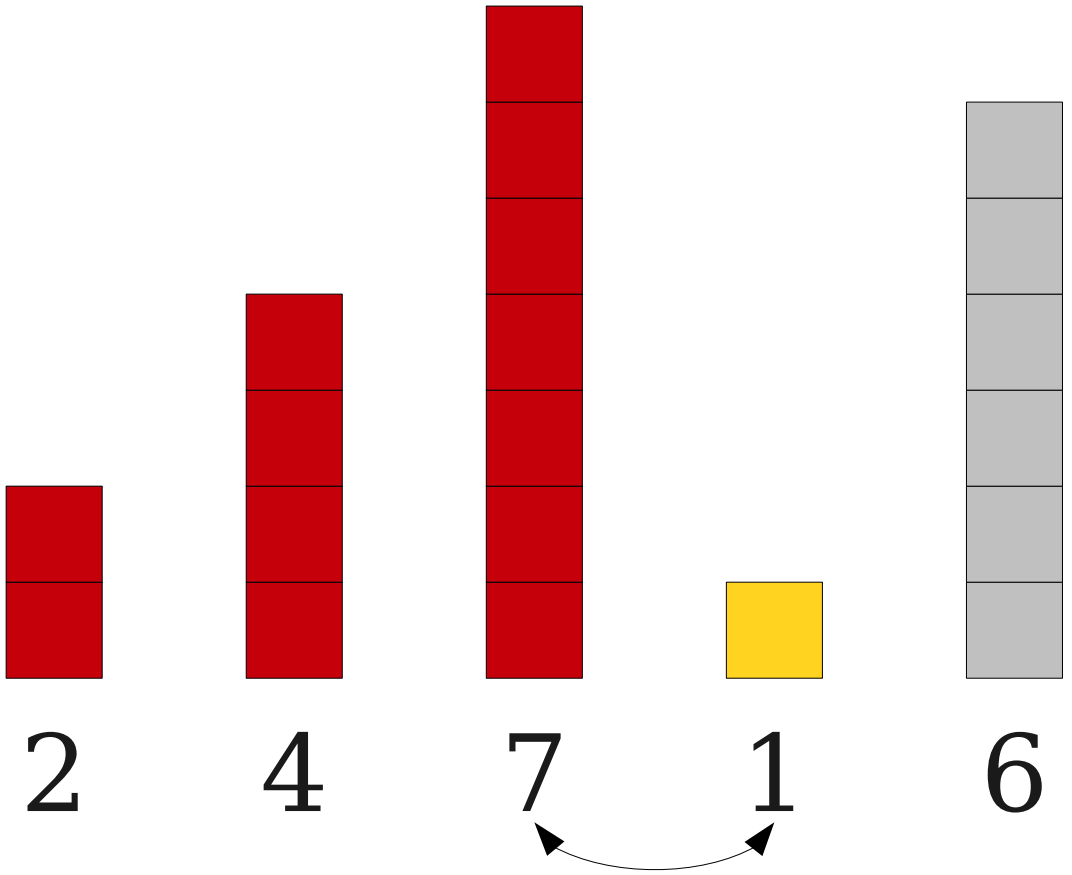


Counting Inversions

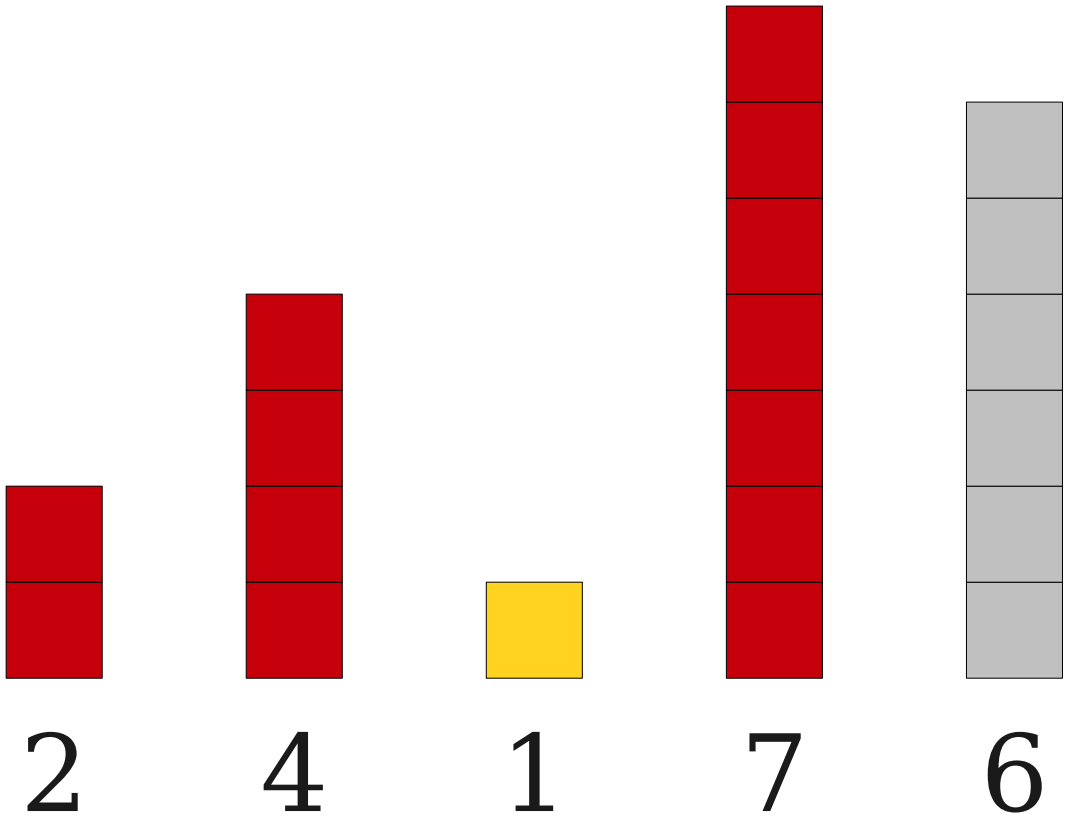


Total Inversions: **4**

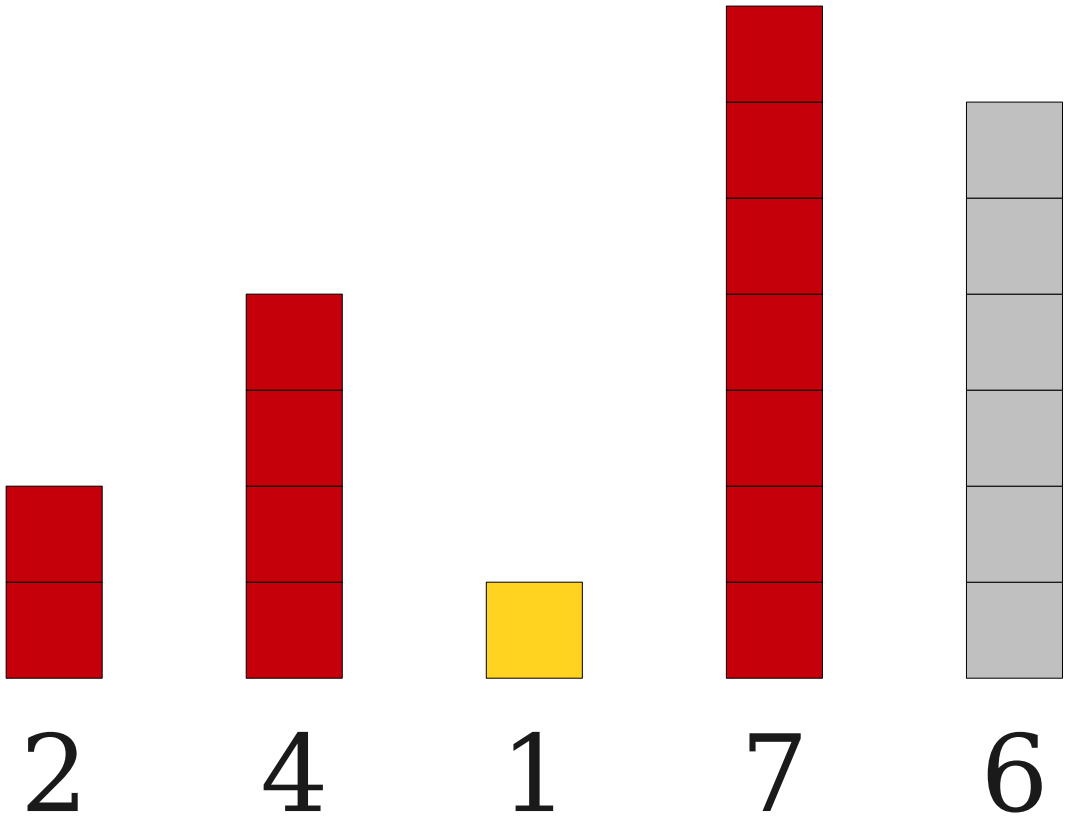
Counting Inversions



Counting Inversions

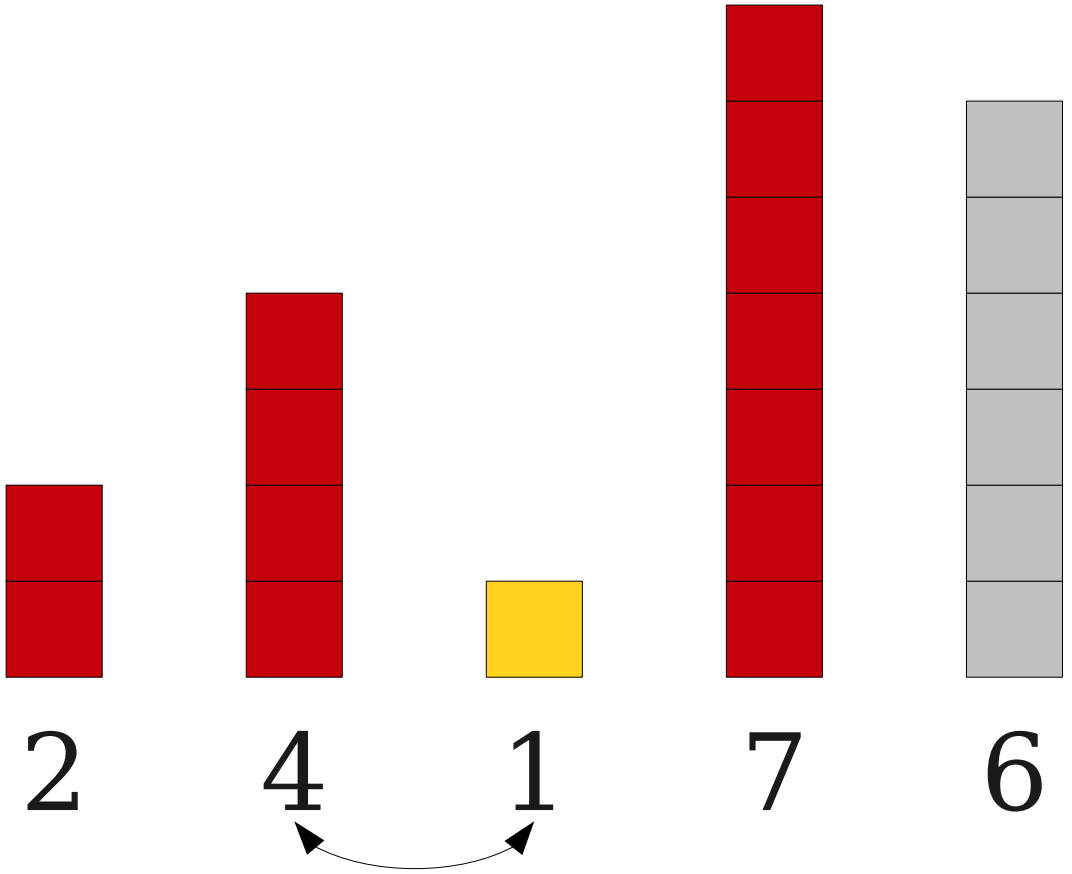


Counting Inversions

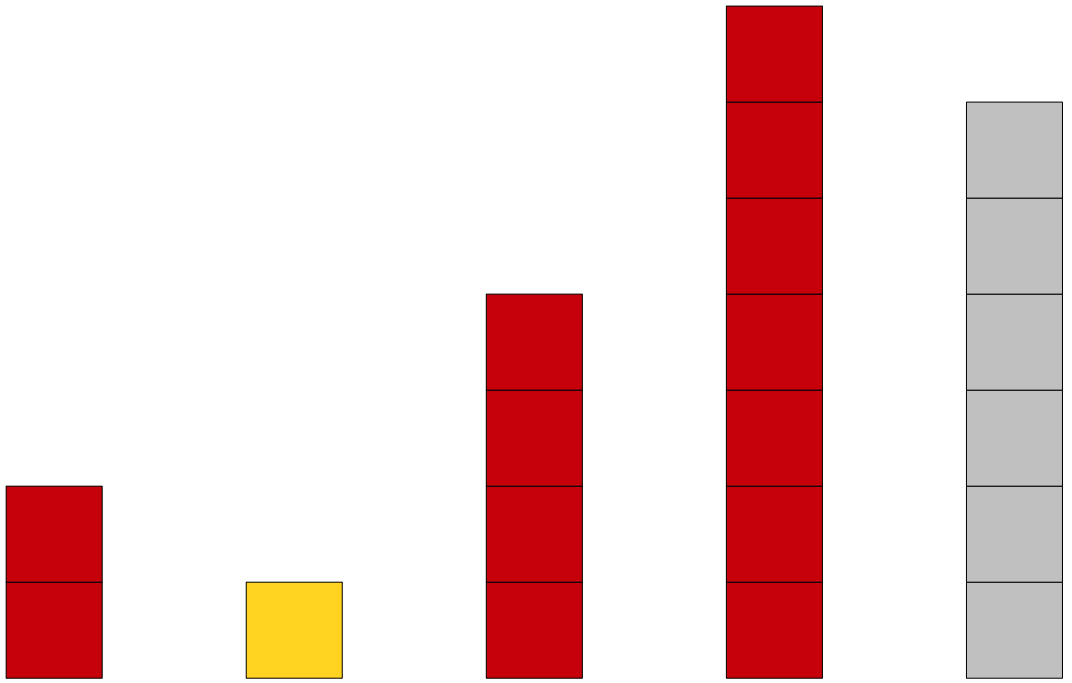


Total Inversions: **3**

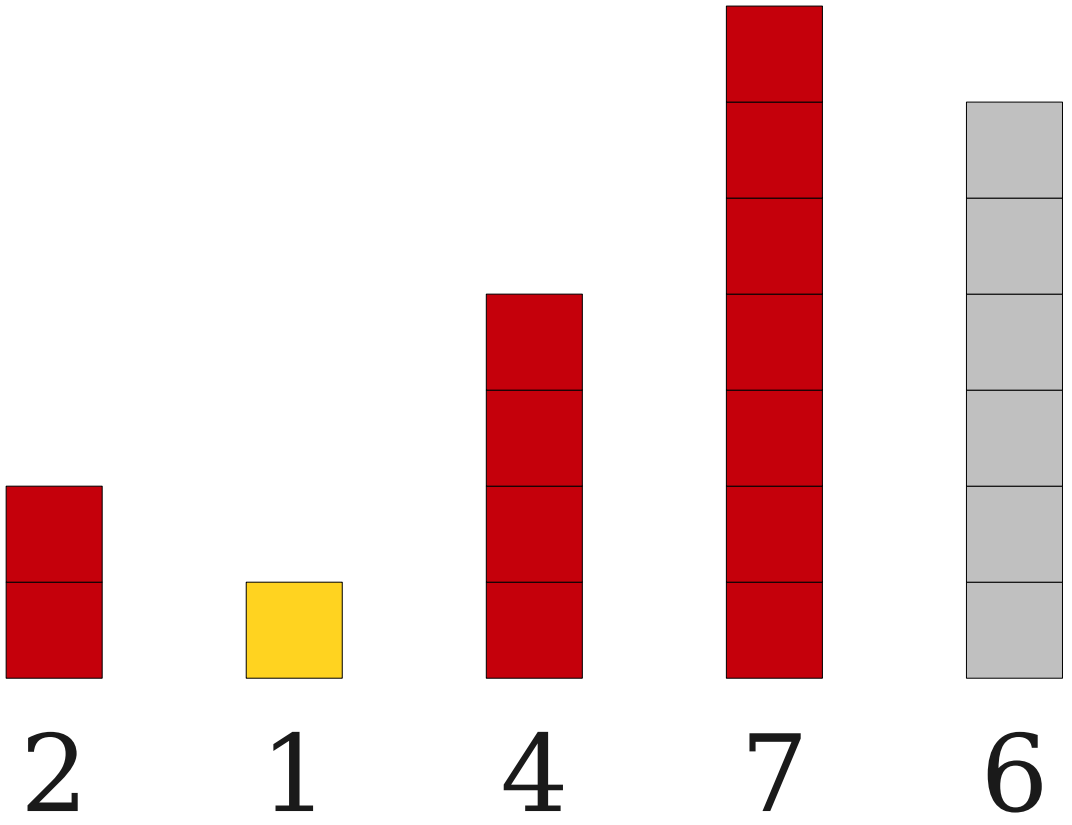
Counting Inversions



Counting Inversions

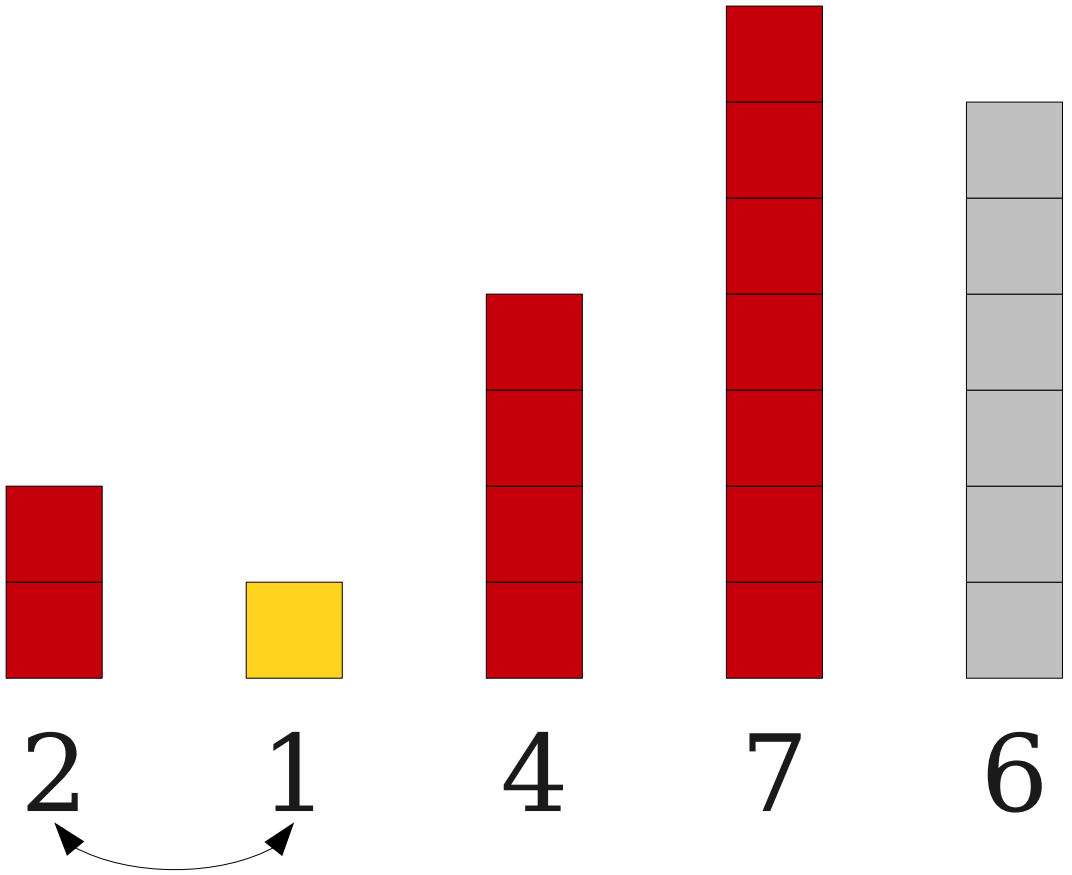


Counting Inversions

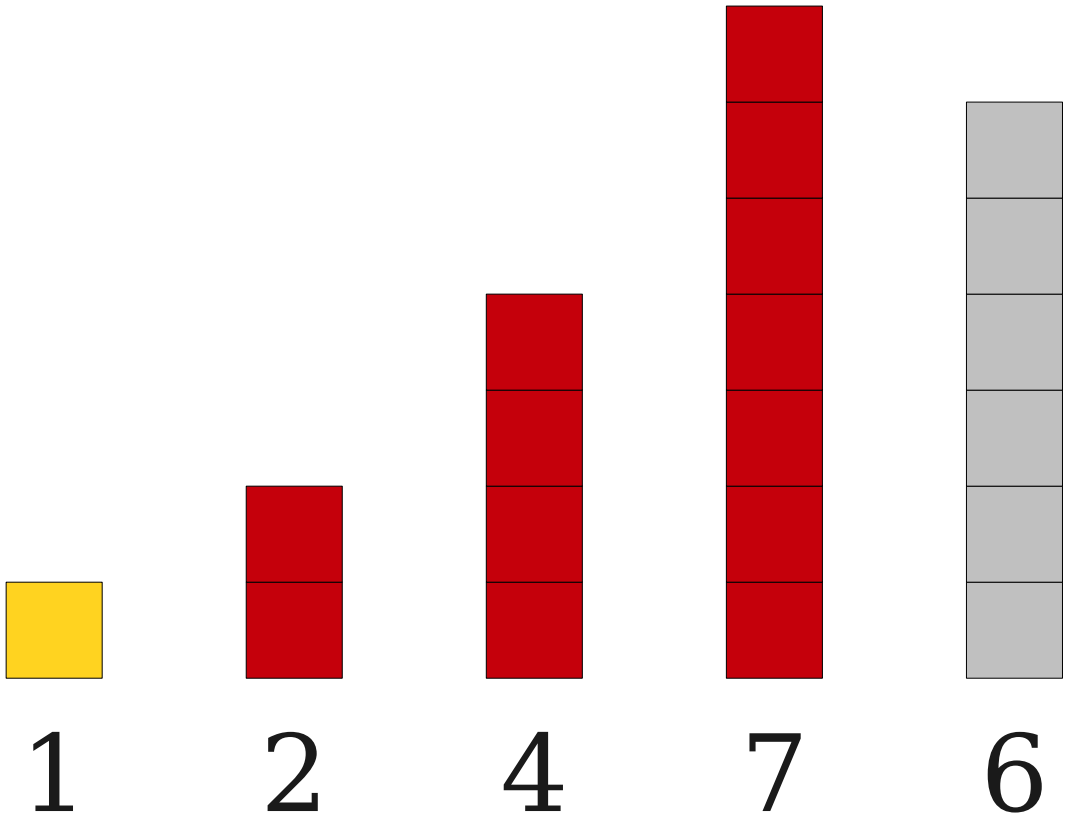


Total Inversions: **2**

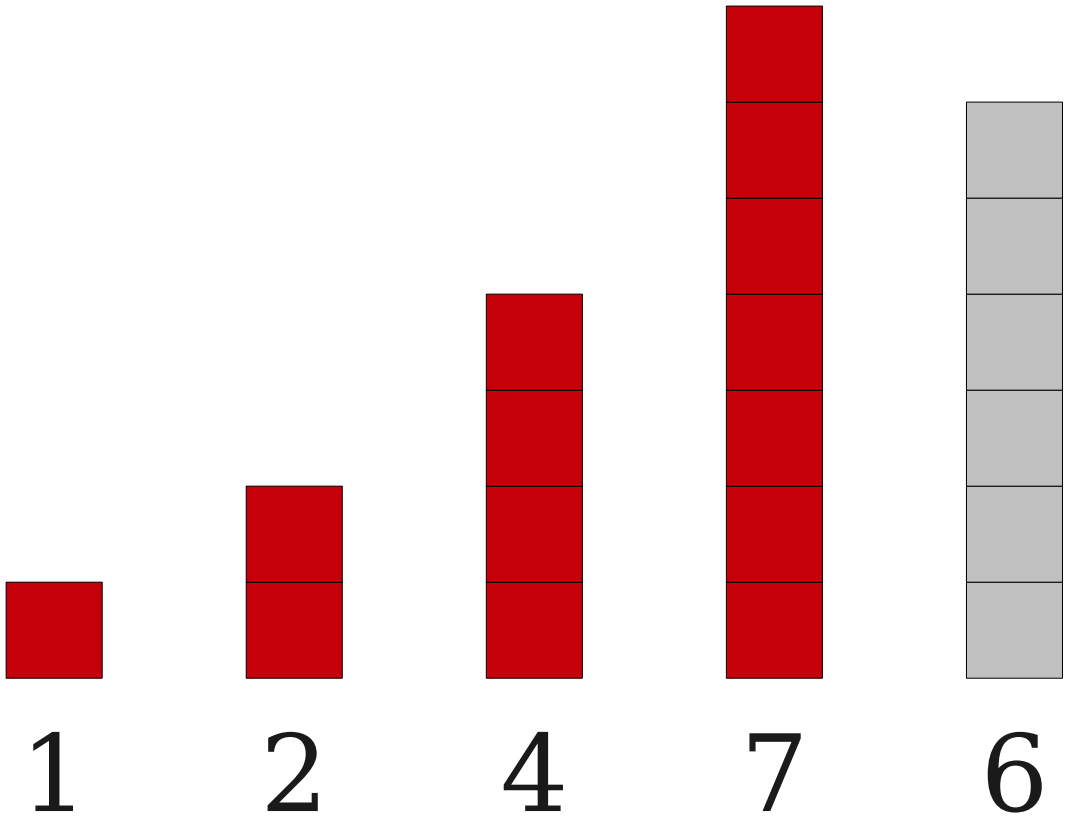
Counting Inversions



Counting Inversions

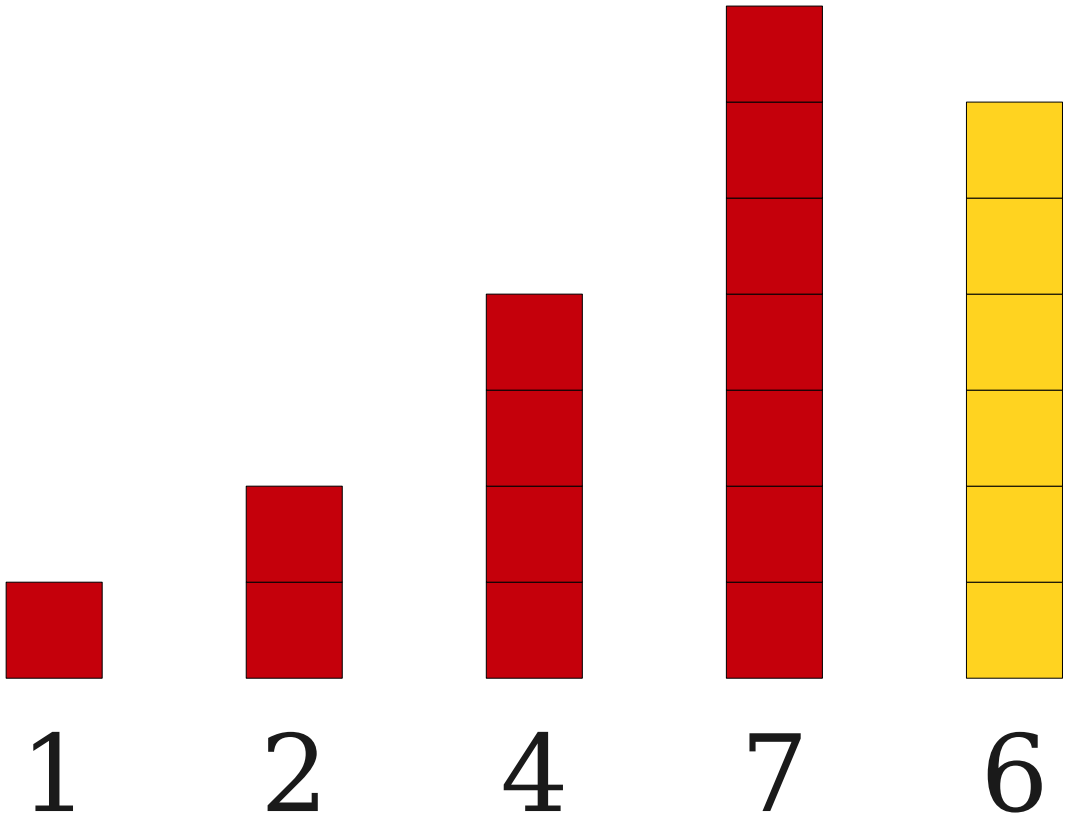


Counting Inversions

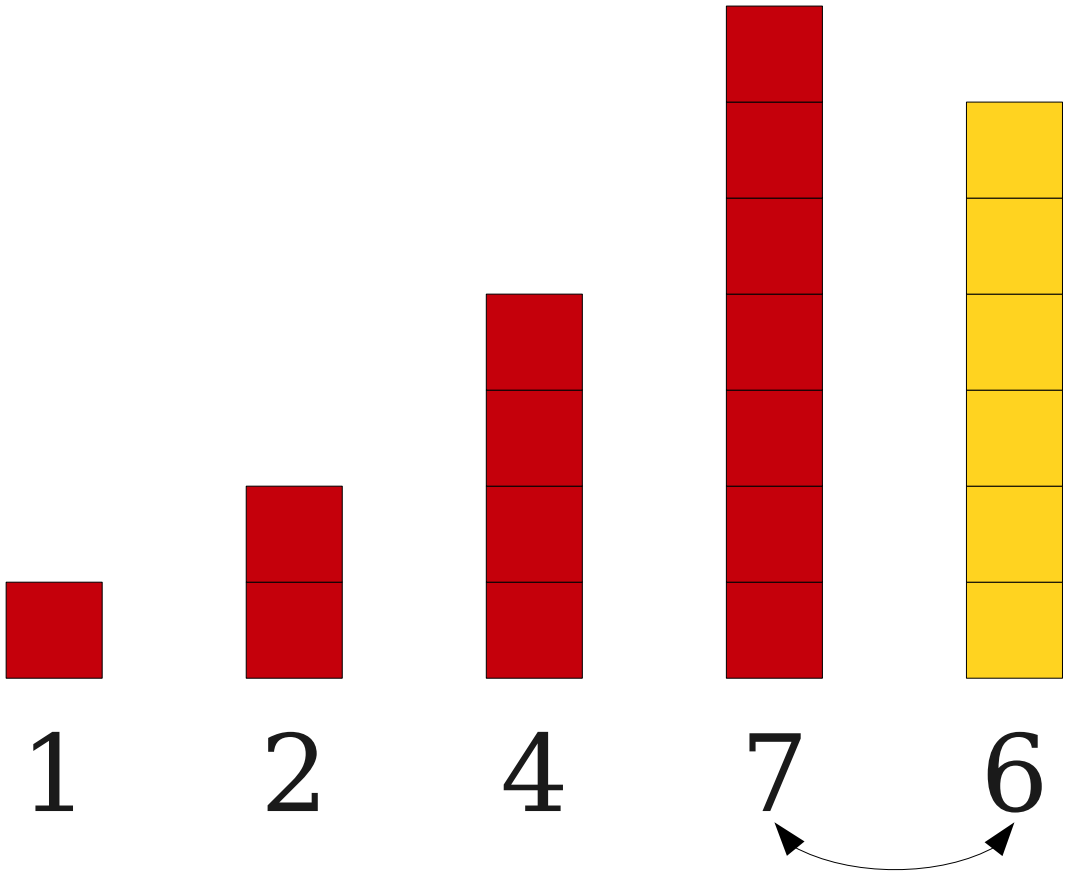


Total Inversions: **1**

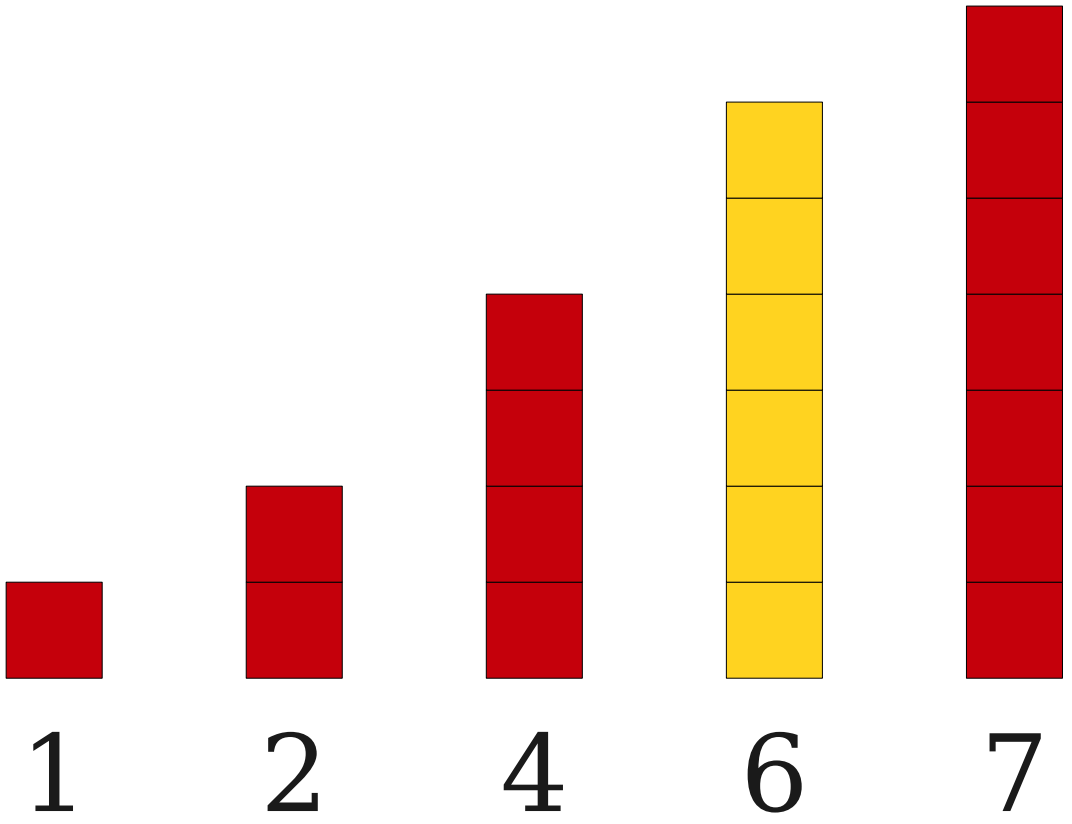
Counting Inversions



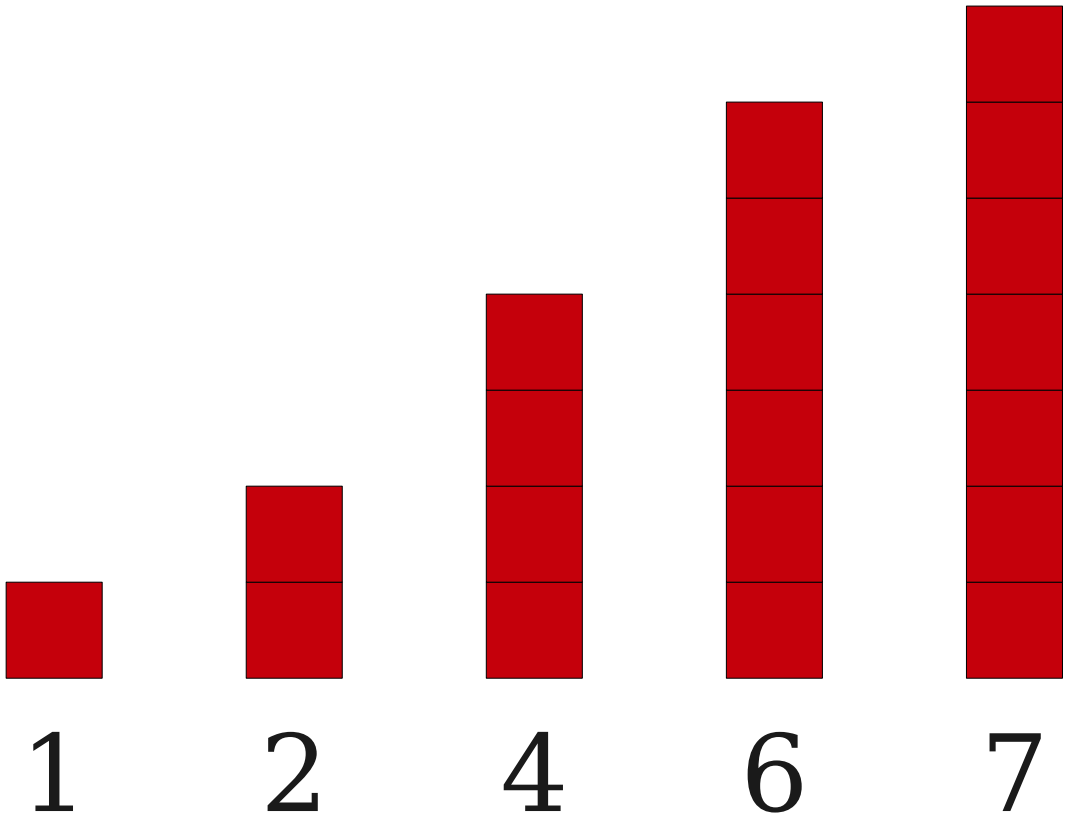
Counting Inversions



Counting Inversions



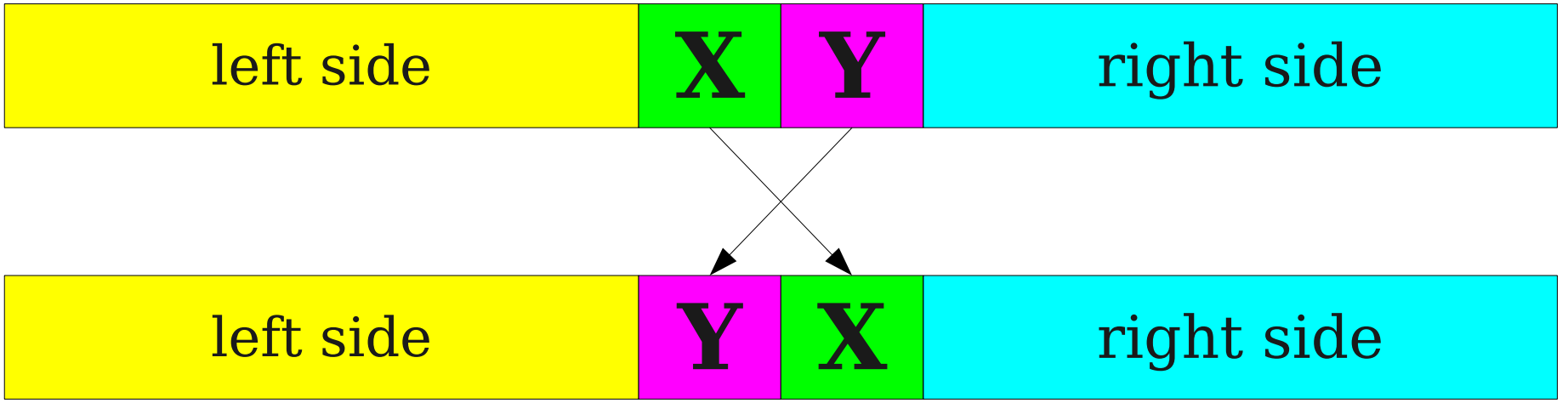
Counting Inversions



Total Inversions: **0**

The Critical Observation

- Every swap performed by insertion sort decreases the number of inversions by exactly one.
- Proof idea:



A Better Analysis

- Insertion sort's runtime is

$$\Theta(n + \#\text{swaps})$$

- Since the number of swaps equals the number of inversions I, the runtime is

$$\Theta(n + I)$$

- How many inversions are there in a random permutation of 1 … n?

Average-Case Analysis

- To count up the expected number of inversions in a random array, we can determine the probability that any individual pair is an inversion, then sum up across all pairs.
  - The math checks out – we'll see why in a few weeks!
- If the input array is chosen truly at random, the probability that any pair of array elements form an inversion is 50%.
  - 50% chance they're in order, 50% chance they're not.
- There are $n(n - 1) / 2$ pairs of elements in an n-element array.
- Average number of inversions: $n(n - 1) / 4 = \mathbf{\Theta(n^2)}$
- Average runtime for insertion sort: $\Theta(n + n^2) = \mathbf{\Theta(n^2)}$.

Why All This Matters

- Insertion sort is a simple algorithm, but by analyzing its correctness and runtime, we explored the following:
  - Loop invariants and correctness proofs.
  - Asymptotic notation: O, Ω, and Θ notations.
  - Techniques for precisely analyzing algorithms.
- As we start to explore more complex algorithms, we will employ these techniques more extensively.

Next Time

- **Fundamental Graph Algorithms**
  - Breadth-First Search.
  - Representing Graphs.
  - Dijkstra's Algorithm.