Algorithmic Analysis and Sorting

Part Two

Announcements

- 3 Handouts on Website
  - Study Skills
  - Practice Midterm
  - Practice Midterm Solutions
- Still working on Reference Sheet…

Big-O Notation

- Ignore *everything* except the dominant growth term, including constant factors.
- Examples:
  - $4n + 4 = \mathbf{O(n)}$
  - $137n + 271 = \mathbf{O(n)}$
  - $n^2 + 1000n + 100000 = \mathbf{O(n^2)}$
  - $2^n + n^3 = \mathbf{O(2^n)}$

Algorithmic Analysis with Big-O

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

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

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, CS365, or CS369N for more information!

Worst Case Analysis

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

   return false;
}
```

- Assume that “ch” is the *worst* possible location for this algorithm
  - In this case, “ch” is not in str

O(n)

Review: Big-O
(Board)

What Can Big-O Tell Us?

- Long-term behavior of a function.
  - If algorithm A is O(n) and algorithm B is $O(n^2)$, **for large inputs** algorithm A will always be faster.
  - If algorithm A is O(n), **for large inputs**, doubling the size of the input roughly doubles the runtime.
    - In other words, Big-O tells us how the running time of an algorithm grows as the size of its input grows

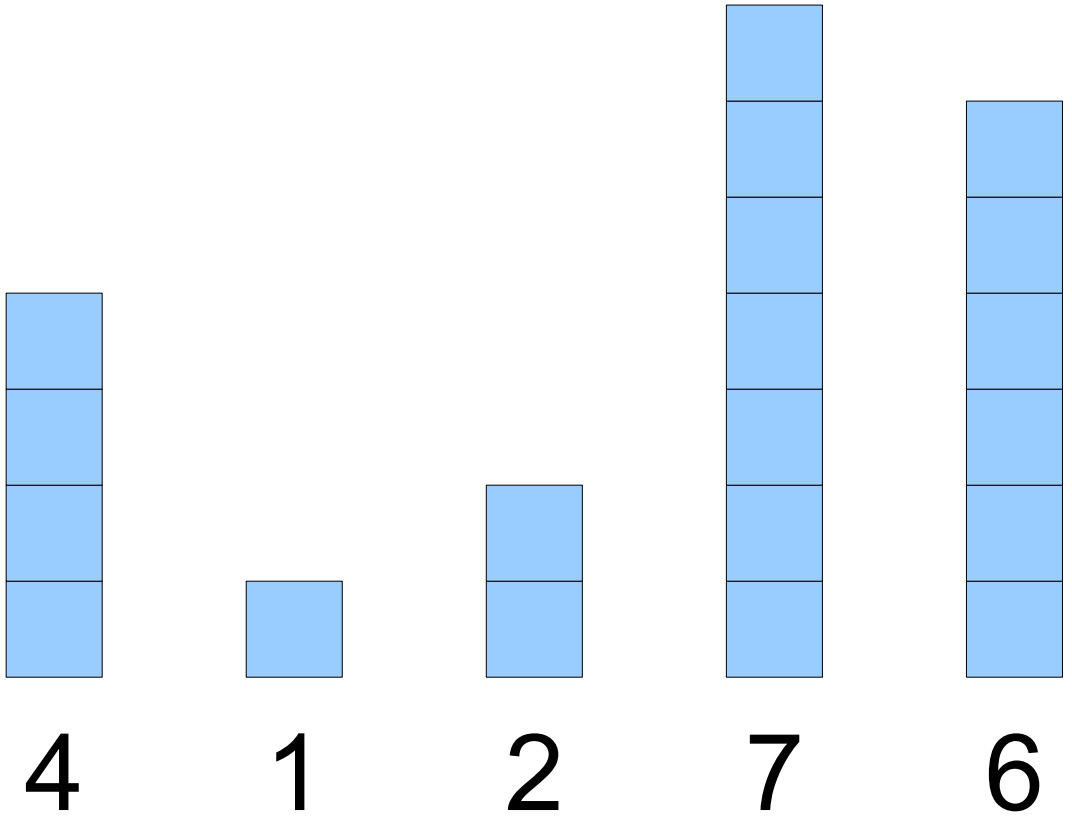
What "large" means on the terms we dropped!

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

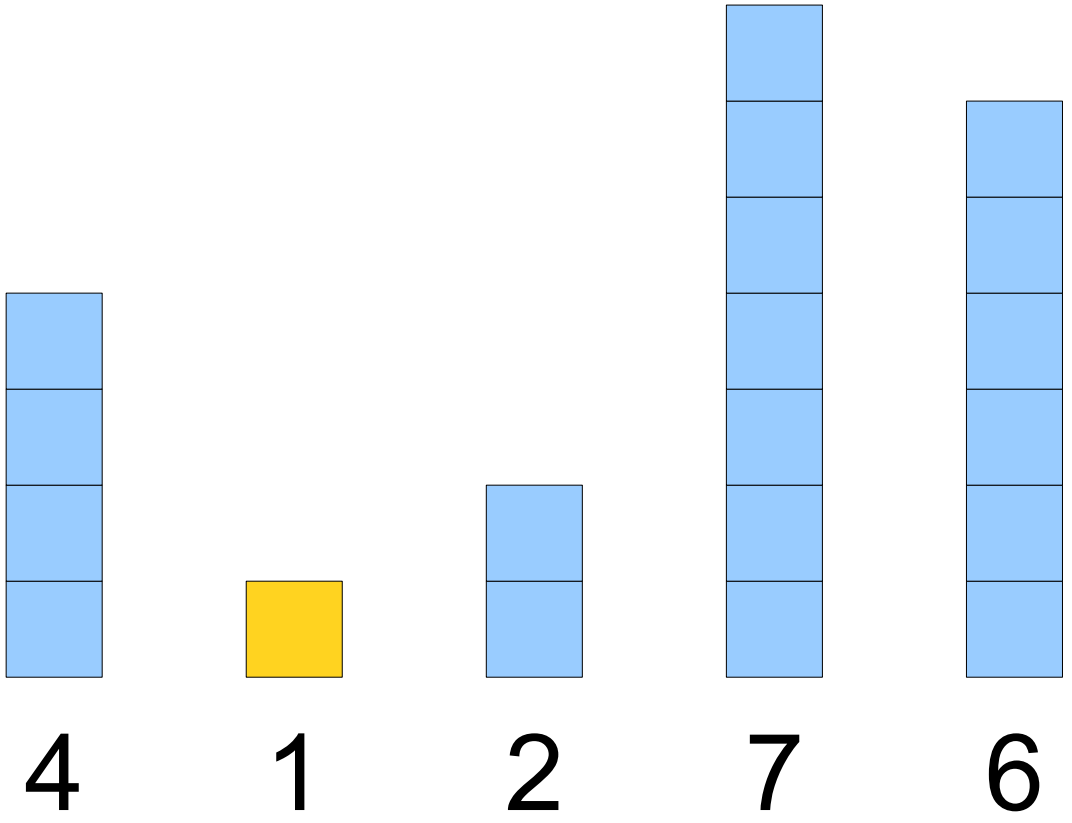
- The actual runtime of a function.
  - $10^{100}n = O(n)$
  - $10^{-100}n = O(n)$
- How a function behaves on small inputs.
  - $n^3 = O(n^3)$
  - $10^6 = O(1)$

An Initial Idea: **Selection Sort**

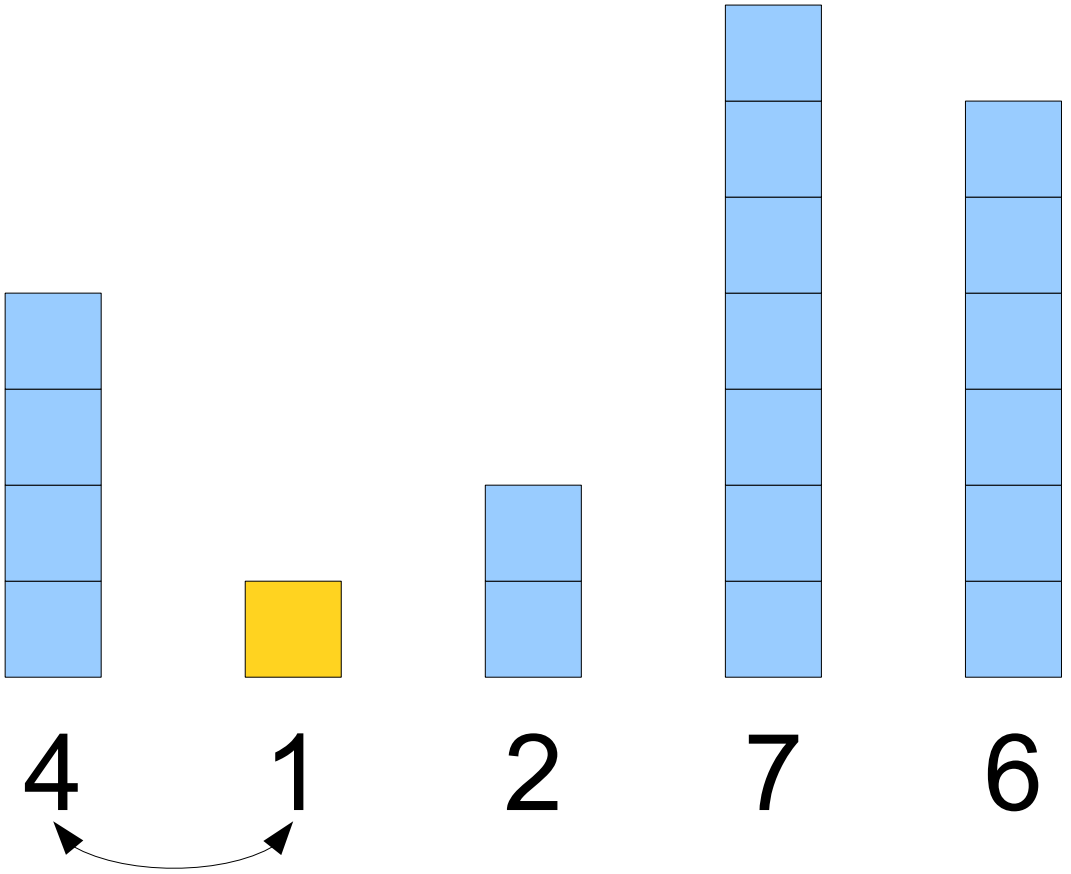
An Initial Idea: **Selection Sort**



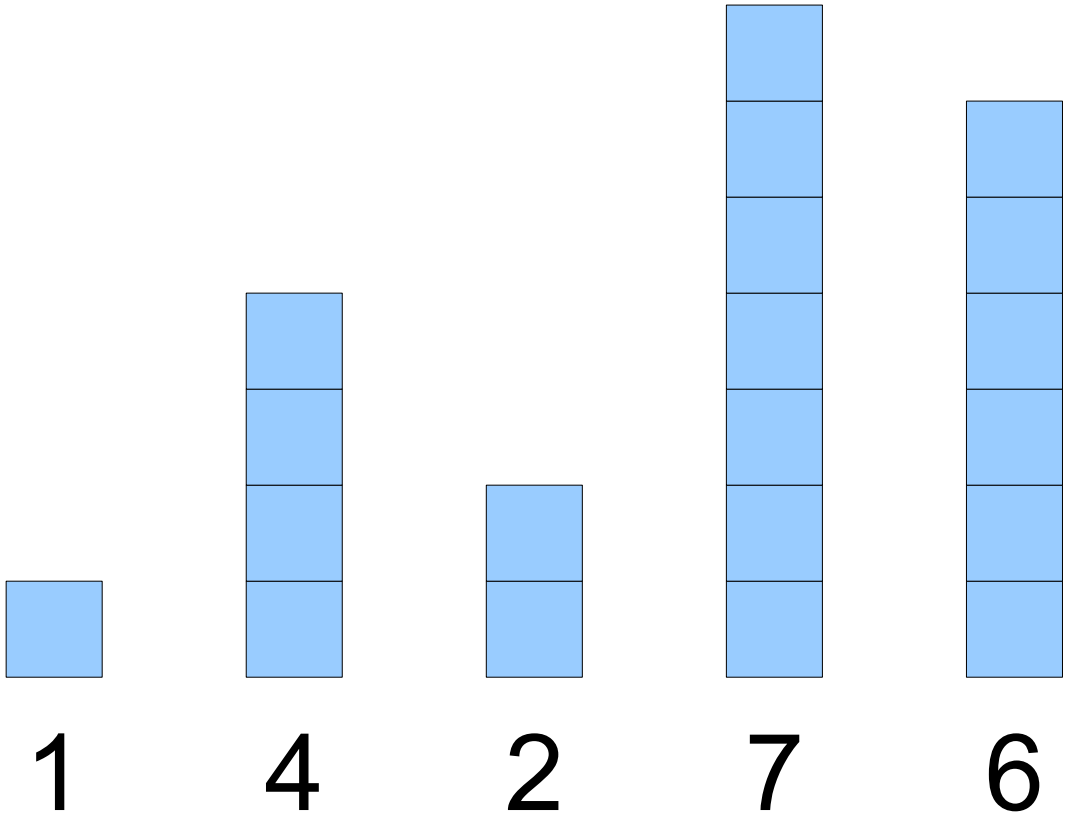
An Initial Idea: **Selection Sort**



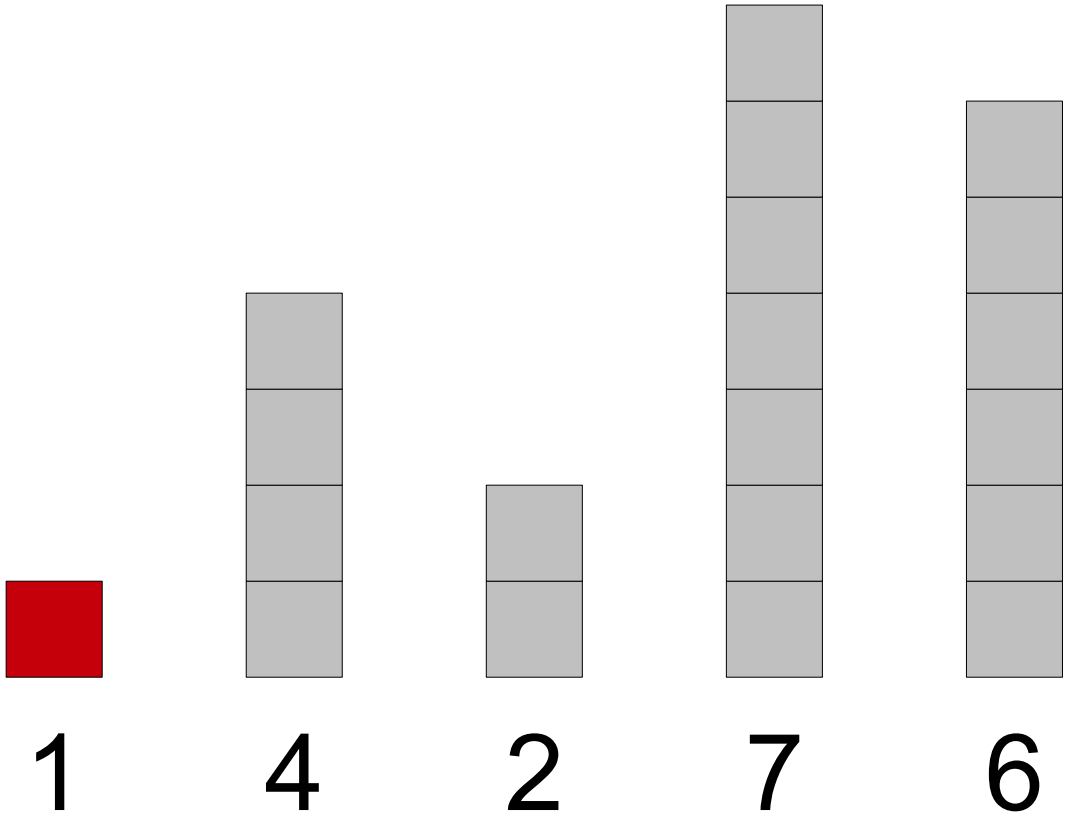
An Initial Idea: **Selection Sort**



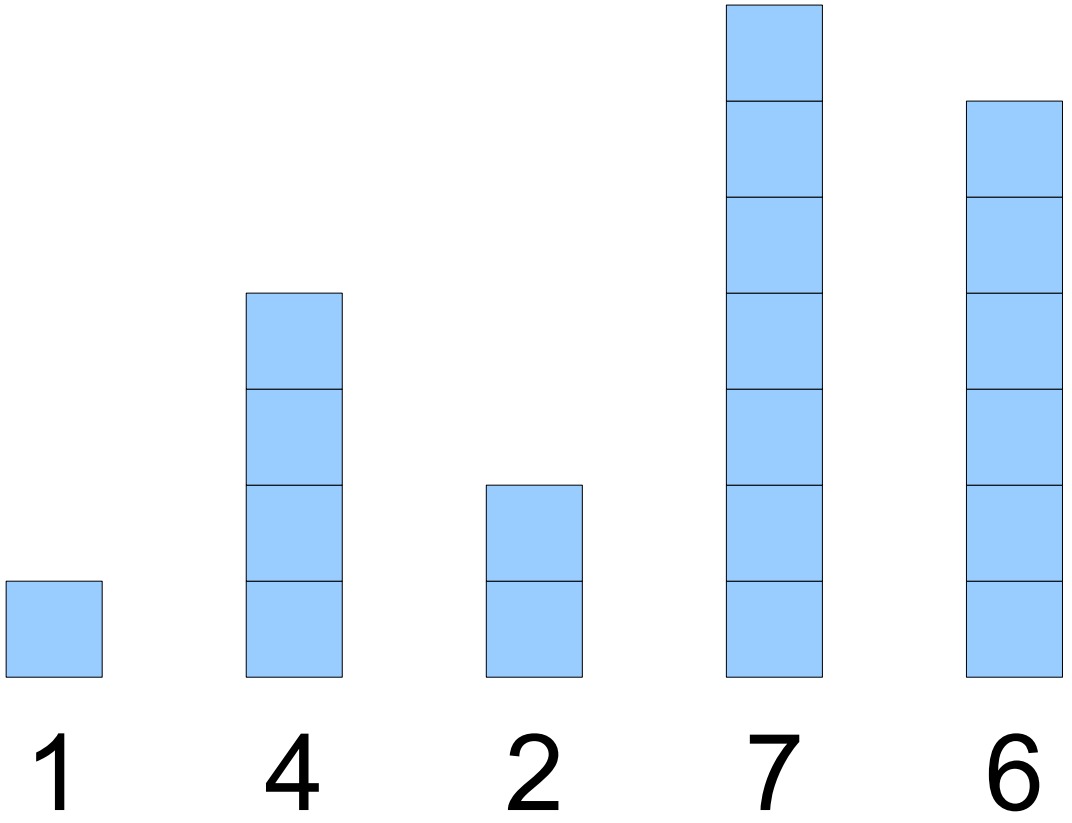
An Initial Idea: **Selection Sort**



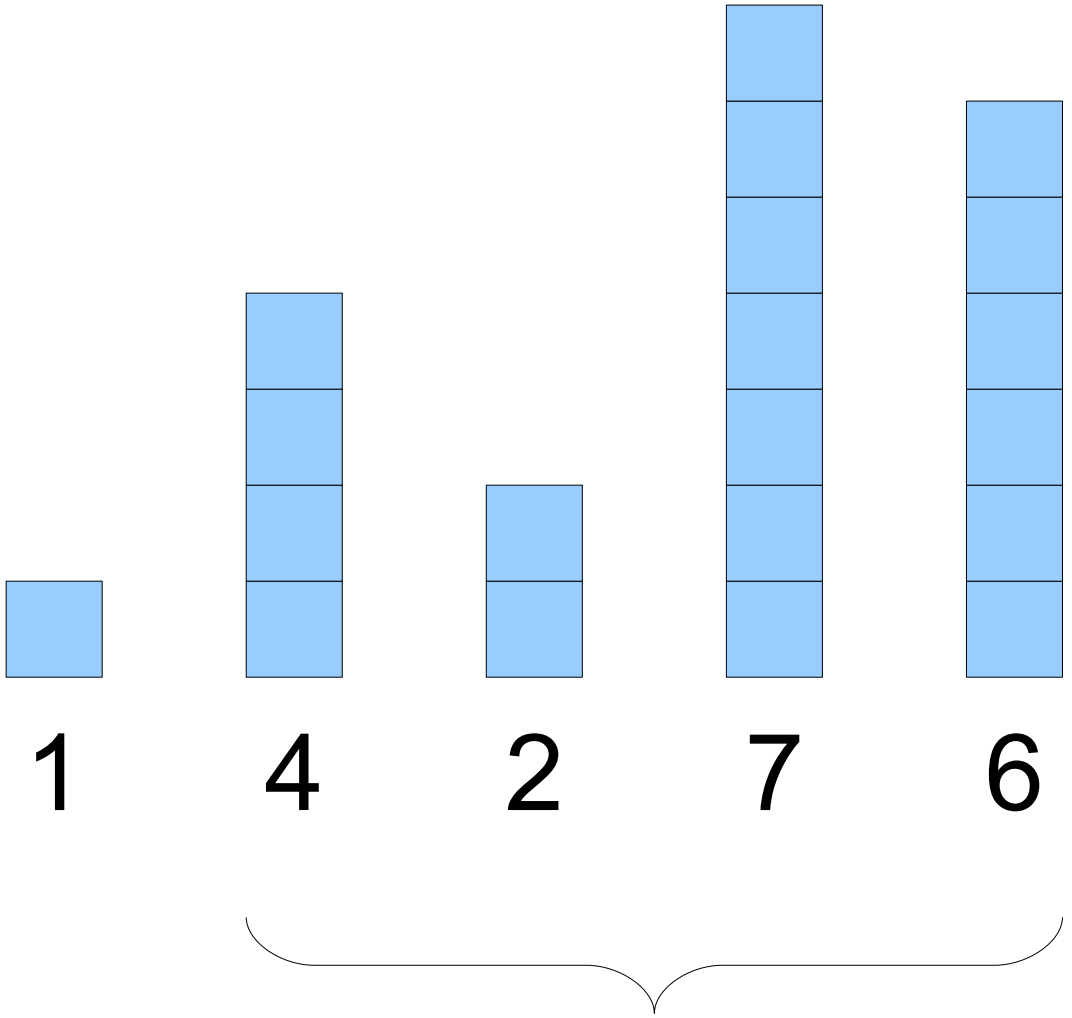
An Initial Idea: **Selection Sort**



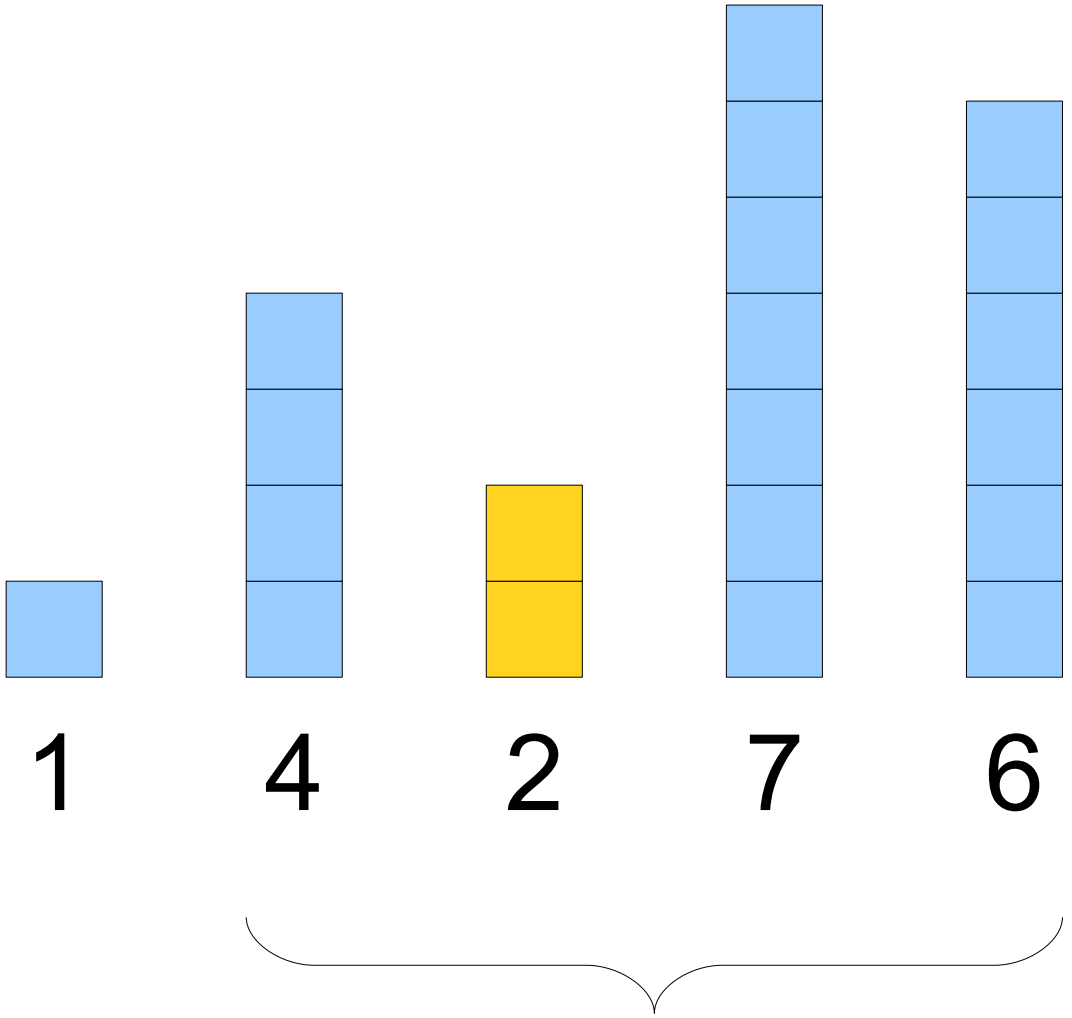
An Initial Idea: **Selection Sort**



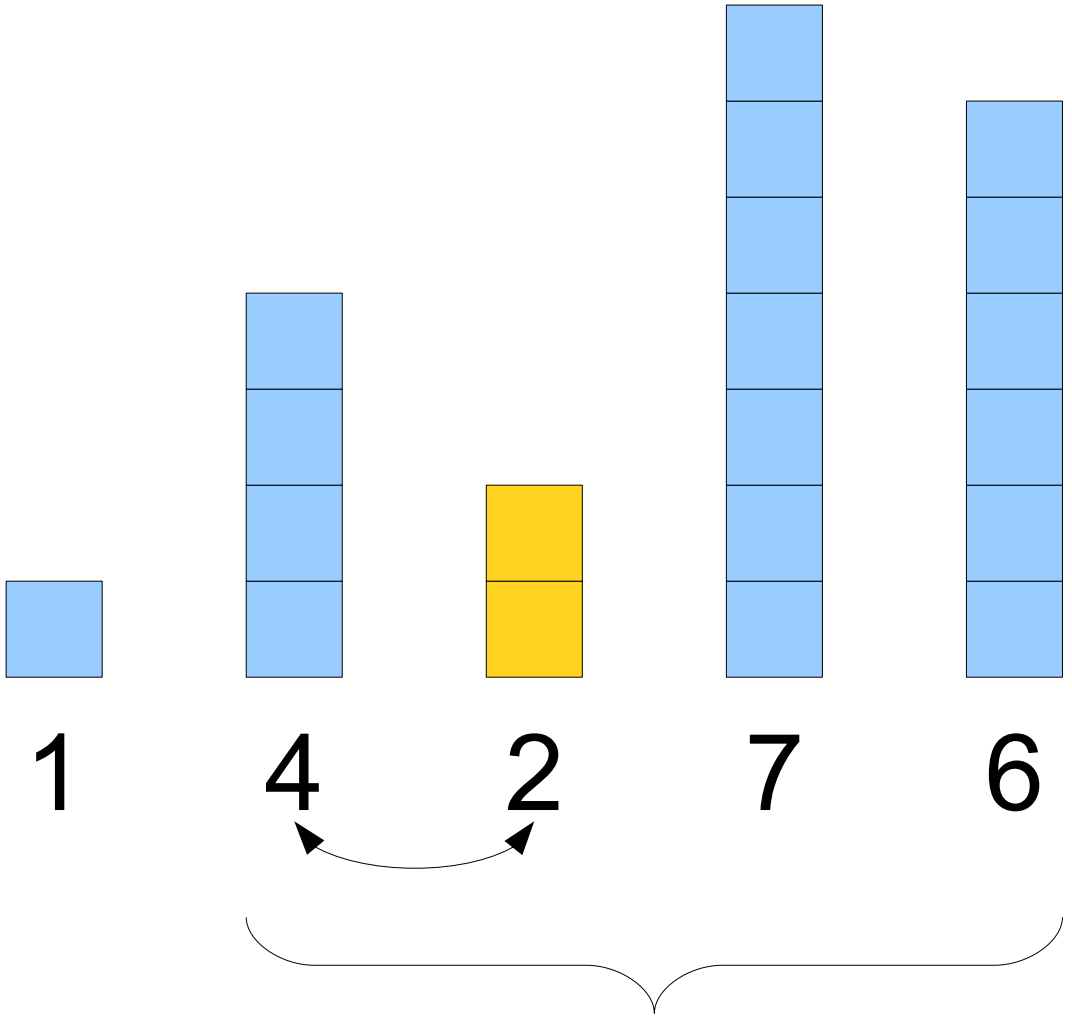
An Initial Idea: **Selection Sort**



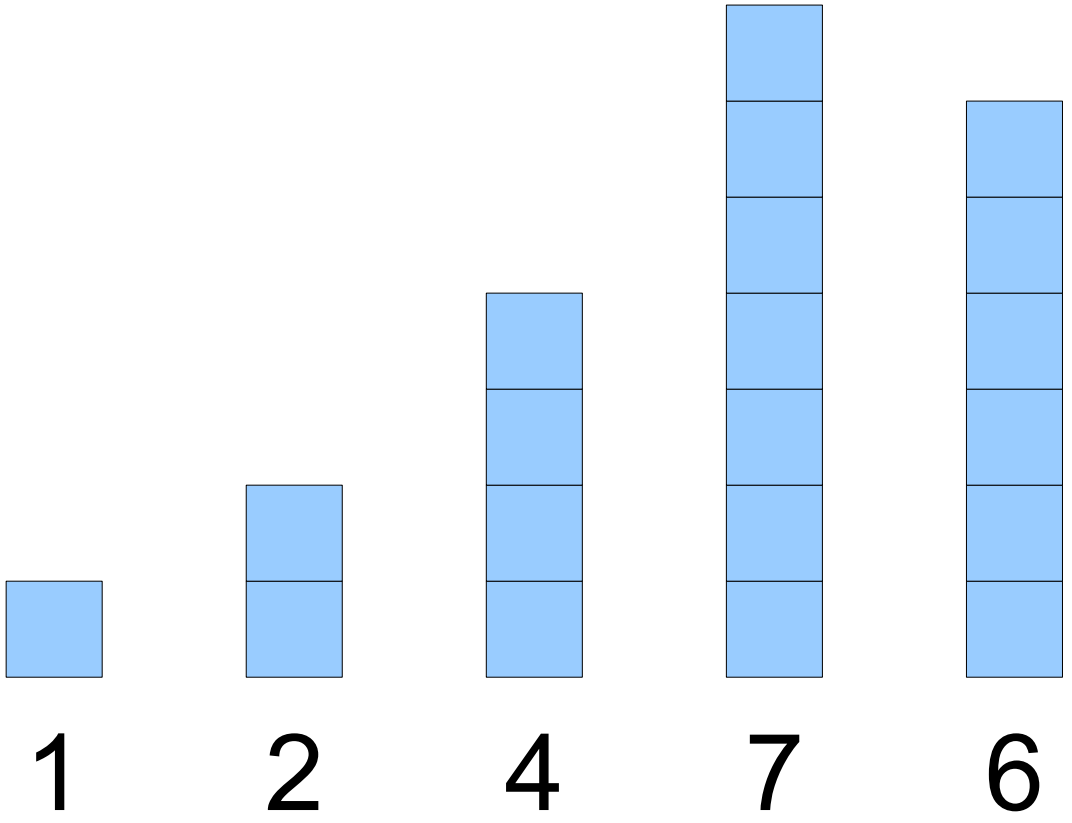
An Initial Idea: **Selection Sort**



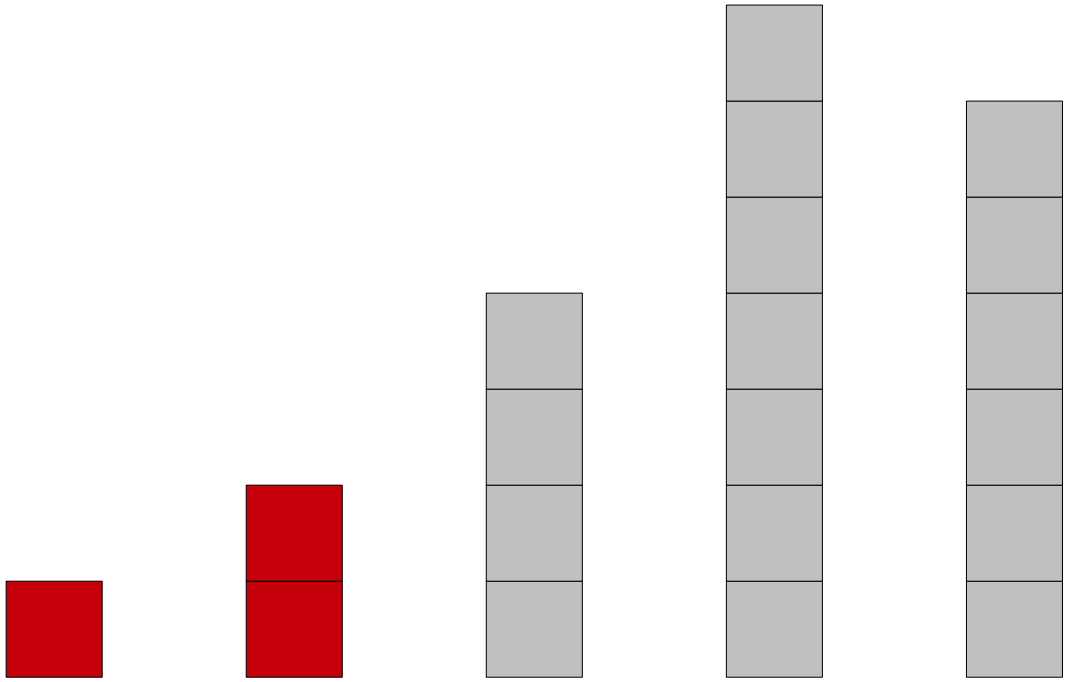
An Initial Idea: **Selection Sort**



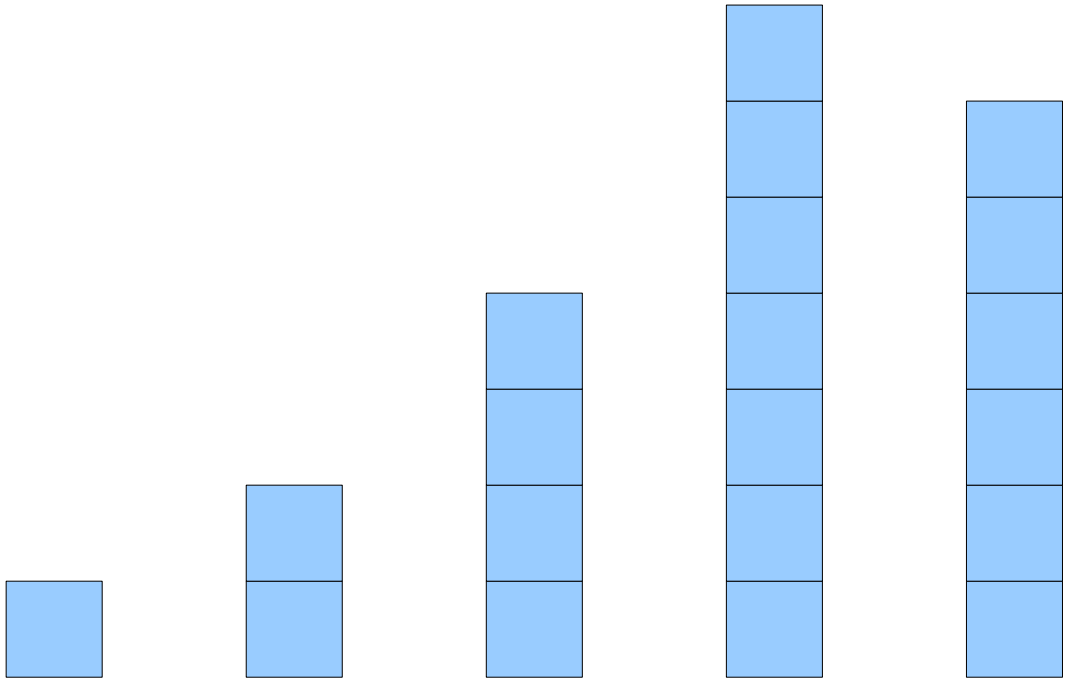
An Initial Idea: **Selection Sort**



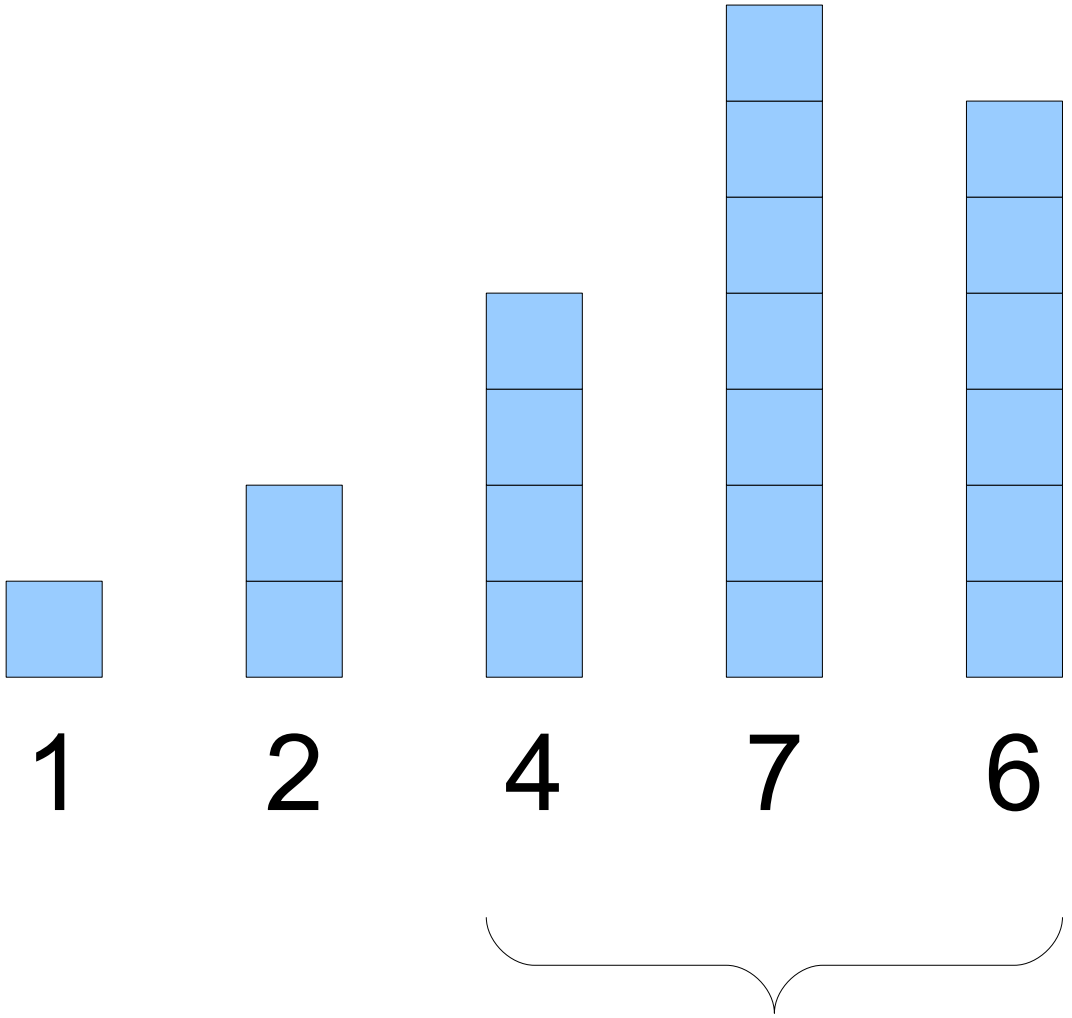
An Initial Idea: **Selection Sort**



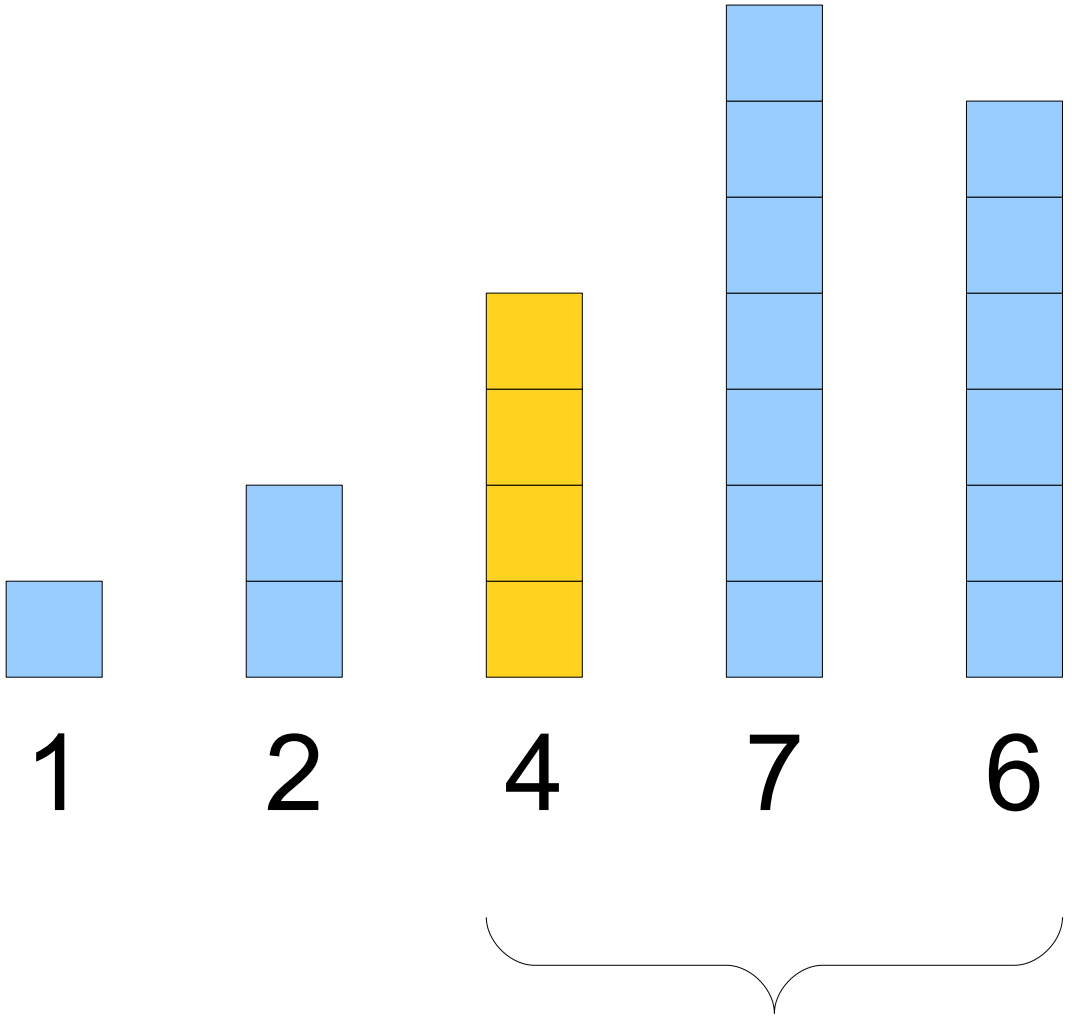
An Initial Idea: **Selection Sort**



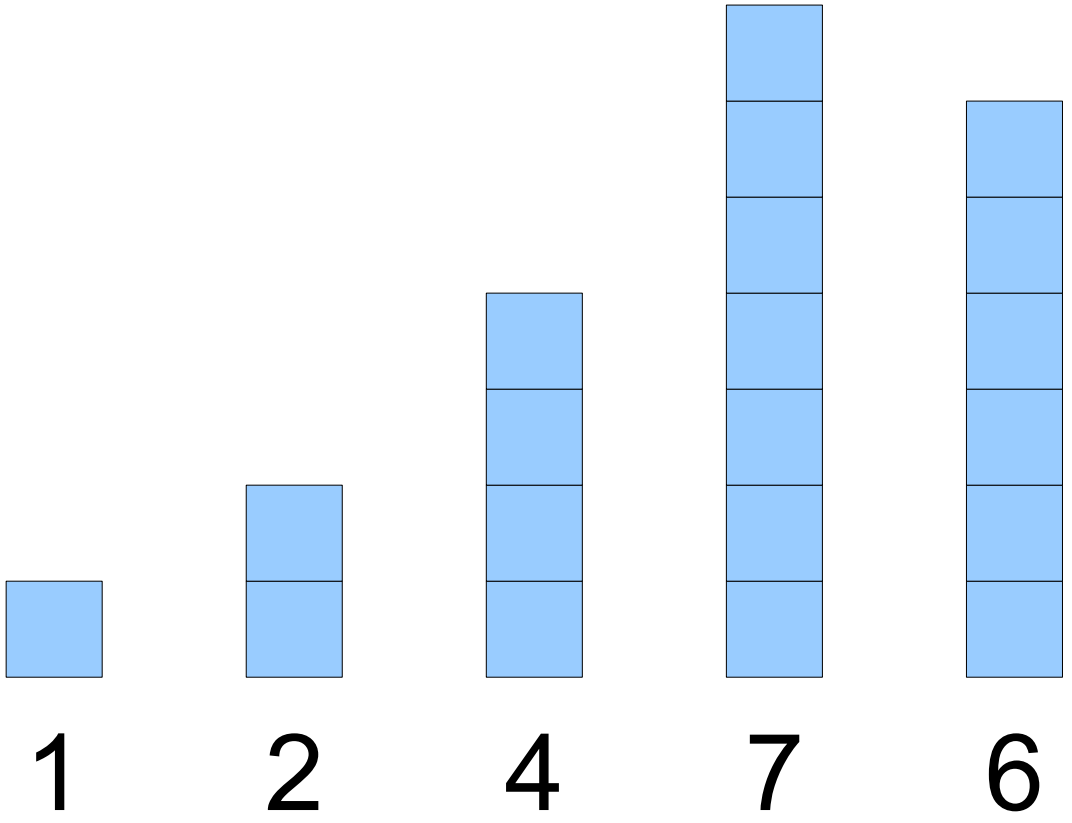
An Initial Idea: **Selection Sort**



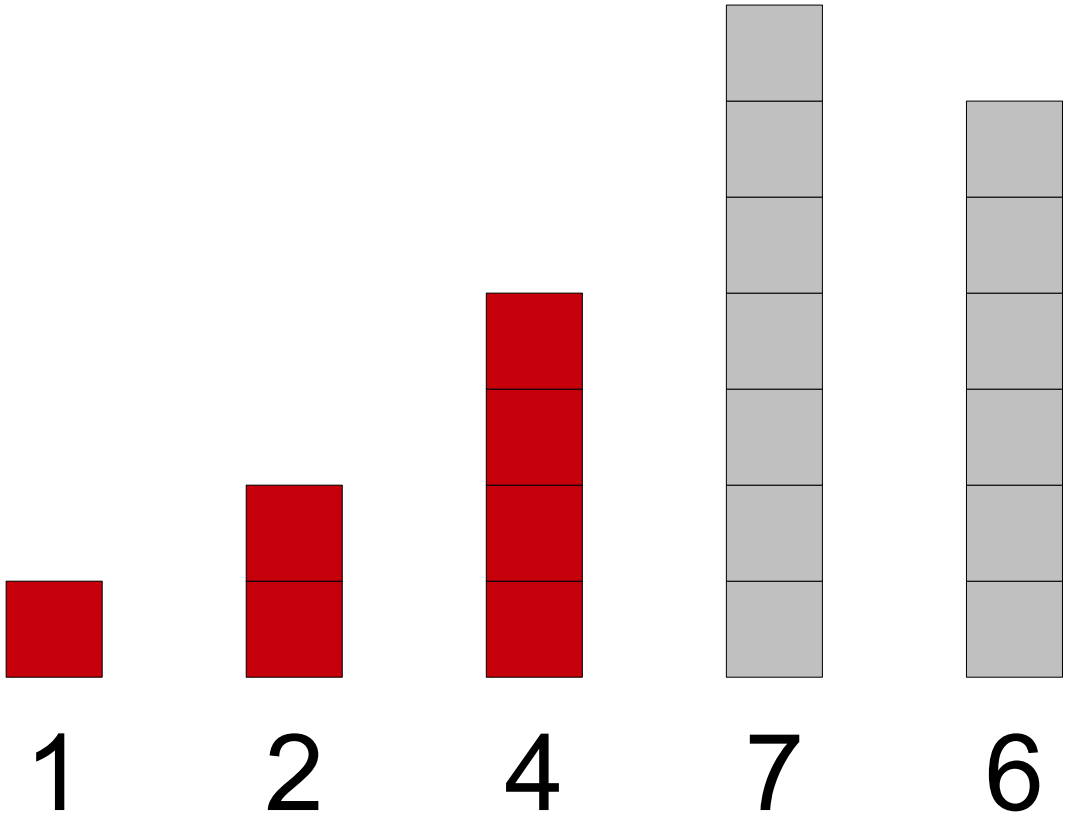
An Initial Idea: **Selection Sort**



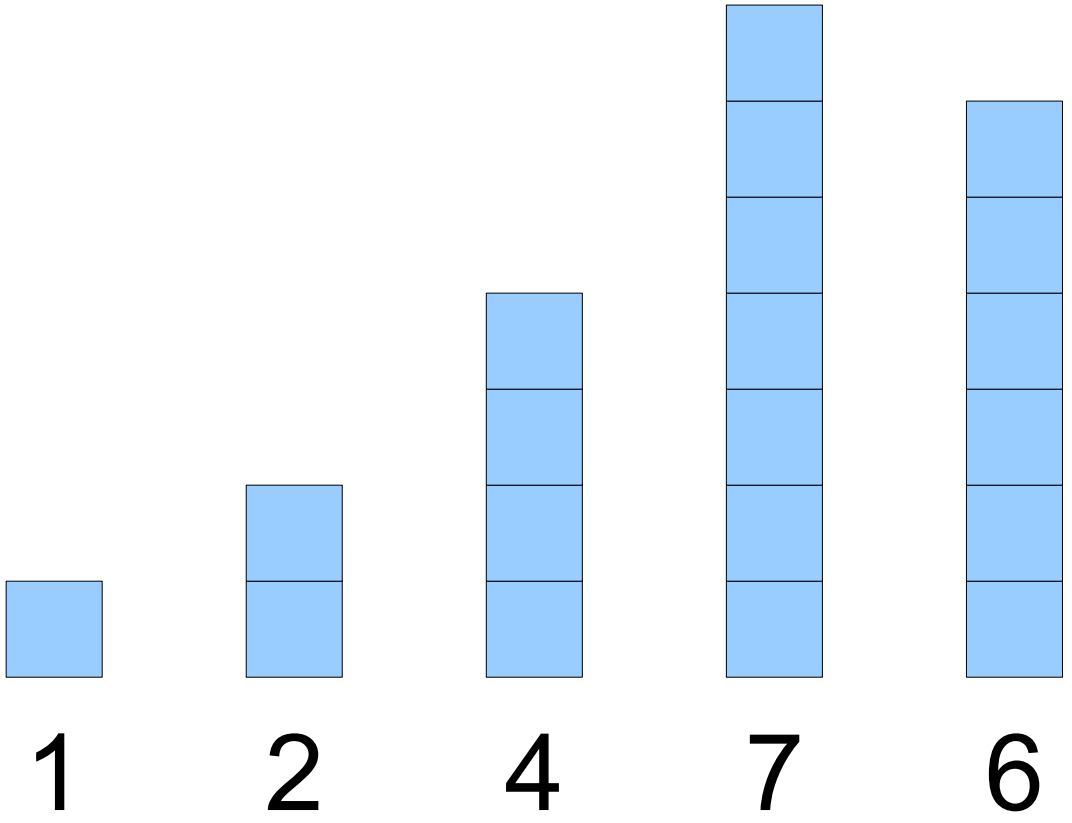
An Initial Idea: **Selection Sort**



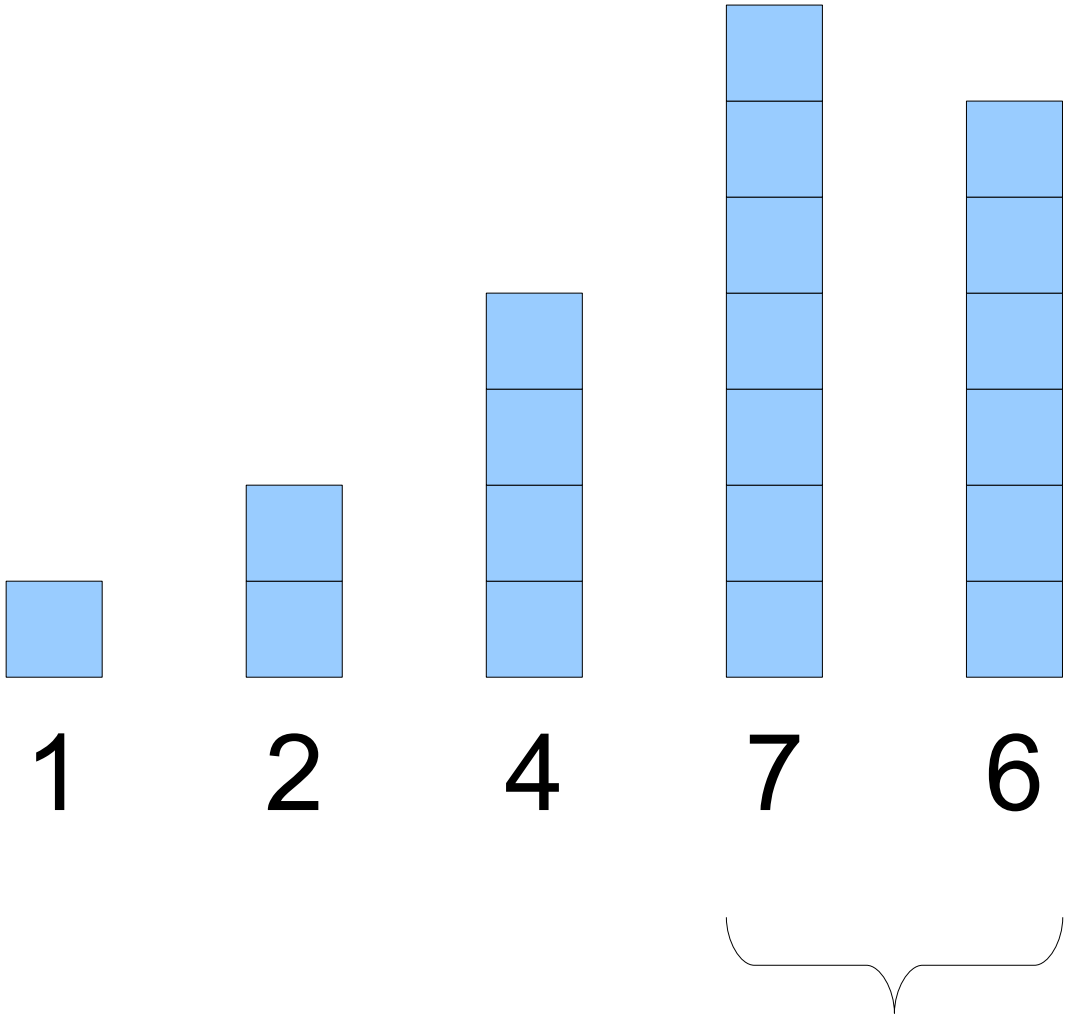
An Initial Idea: **Selection Sort**



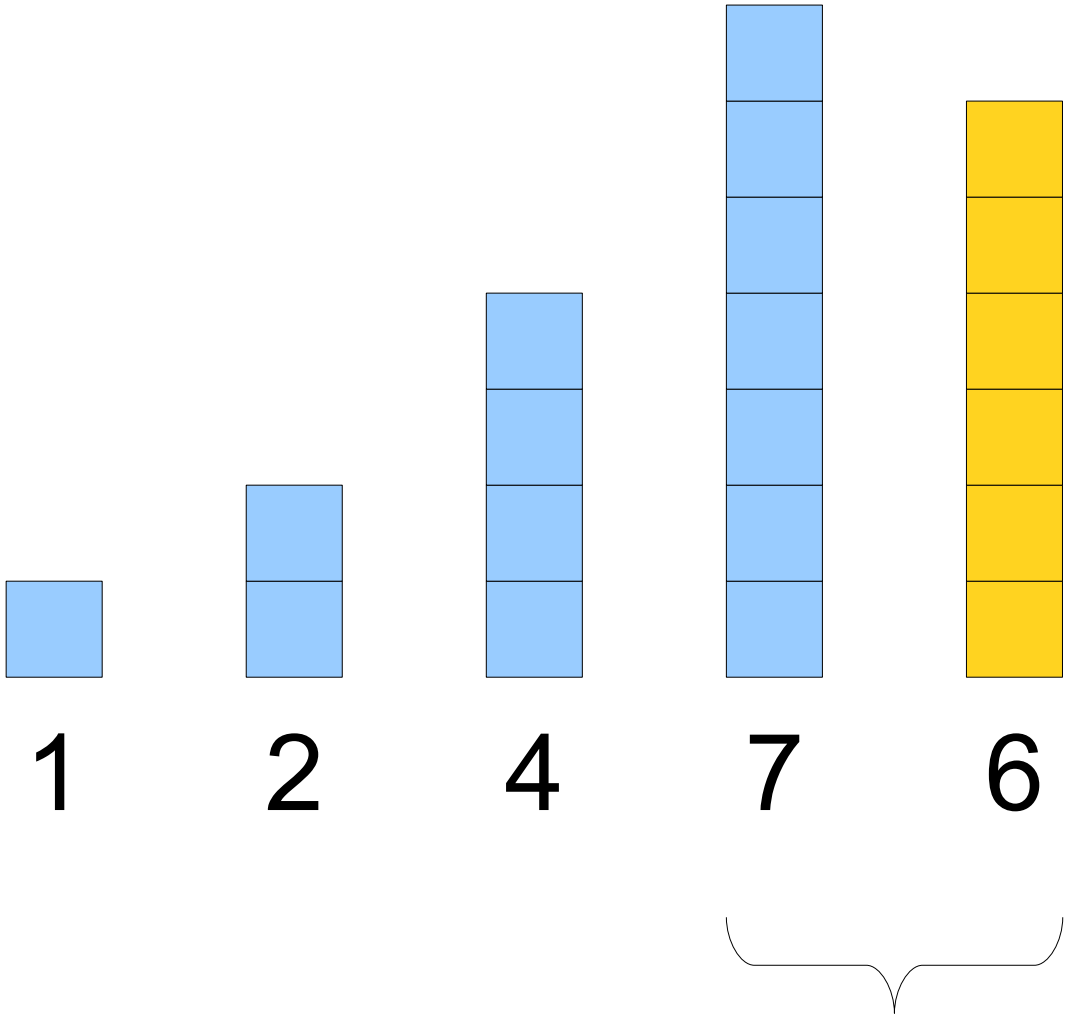
An Initial Idea: **Selection Sort**



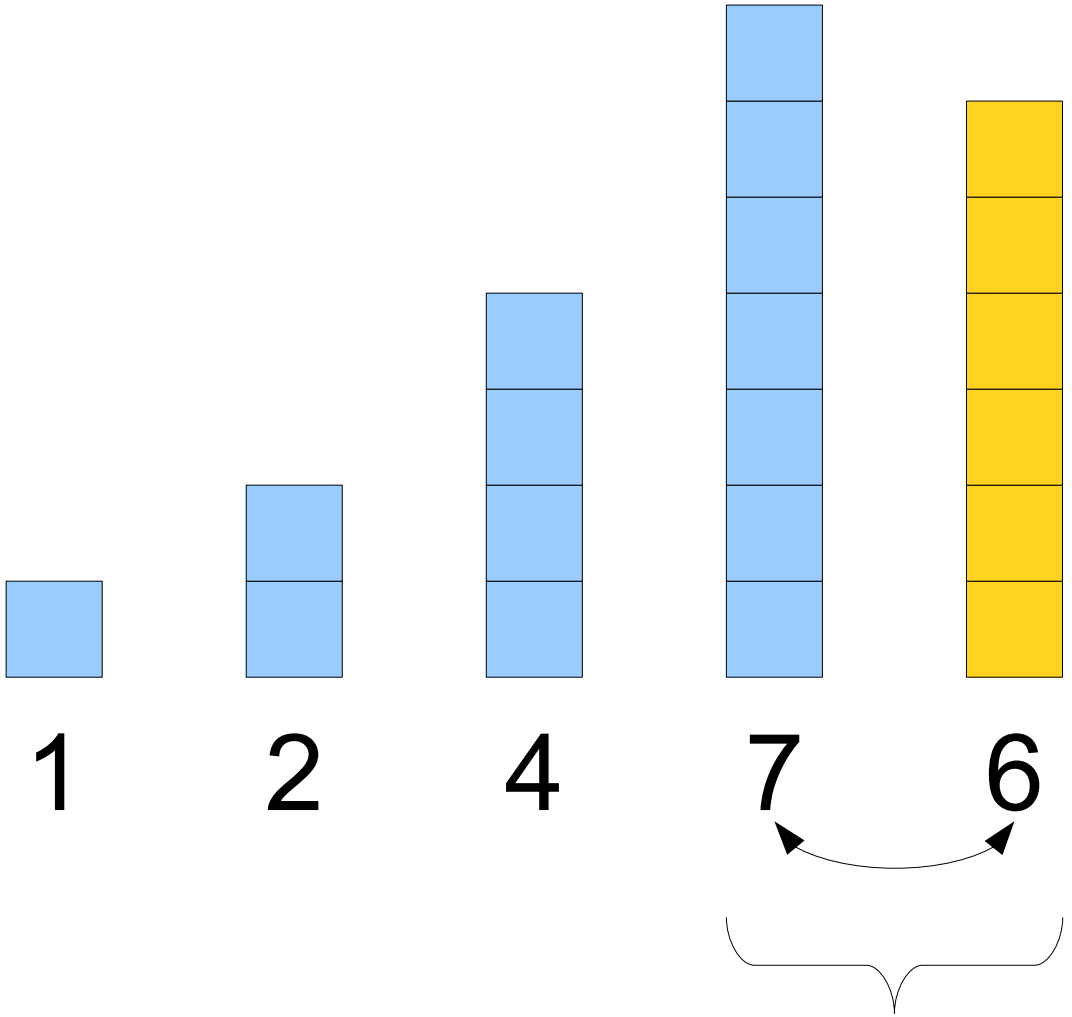
An Initial Idea: **Selection Sort**



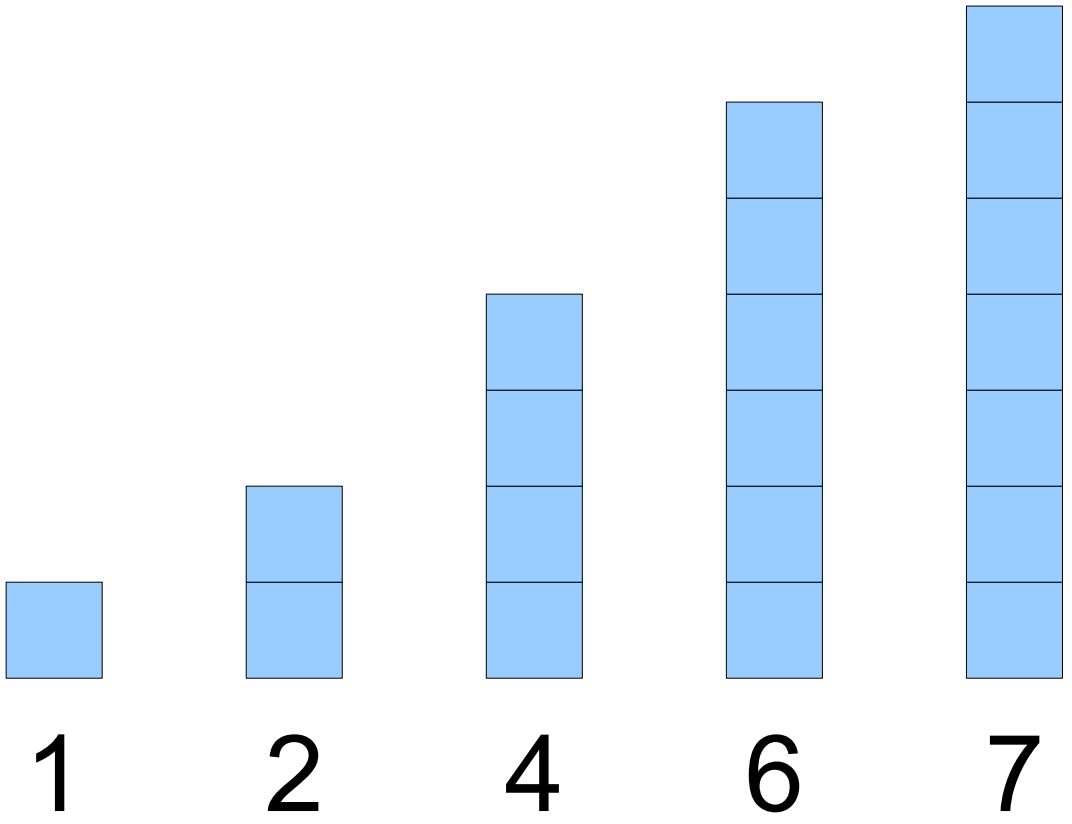
An Initial Idea: **Selection Sort**



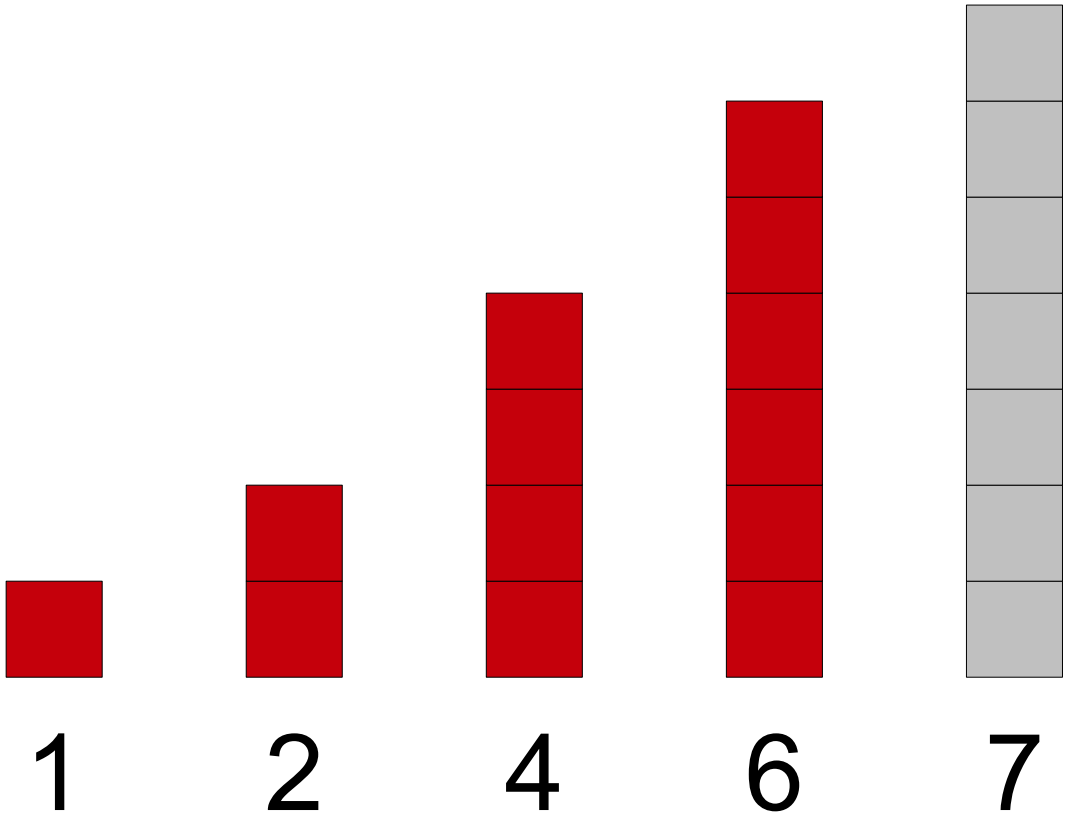
An Initial Idea: **Selection Sort**



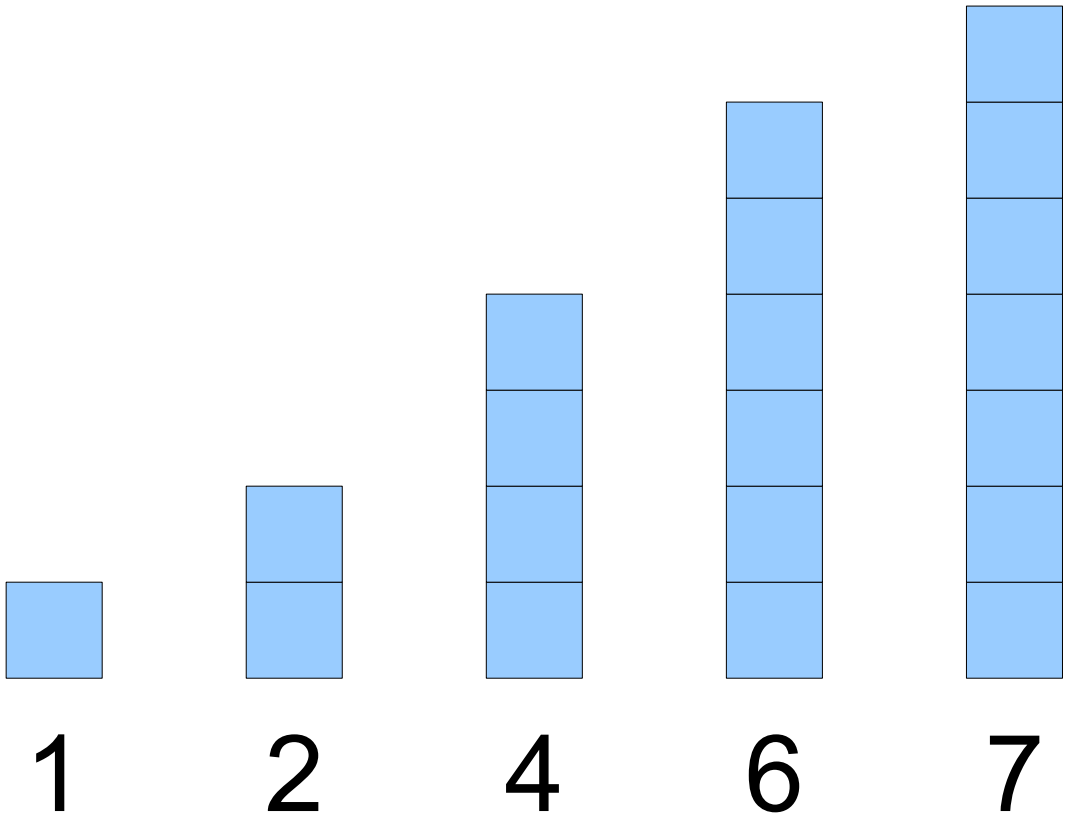
An Initial Idea: **Selection Sort**



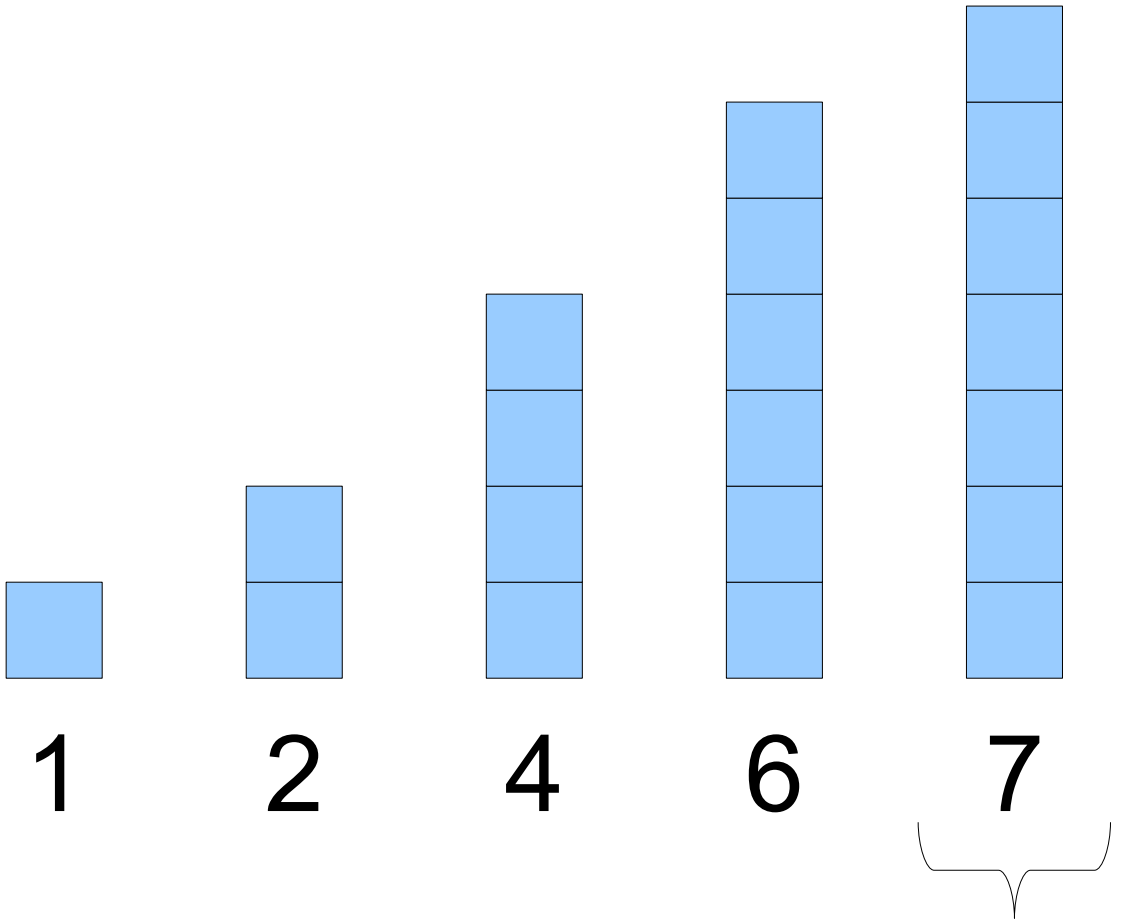
An Initial Idea: **Selection Sort**



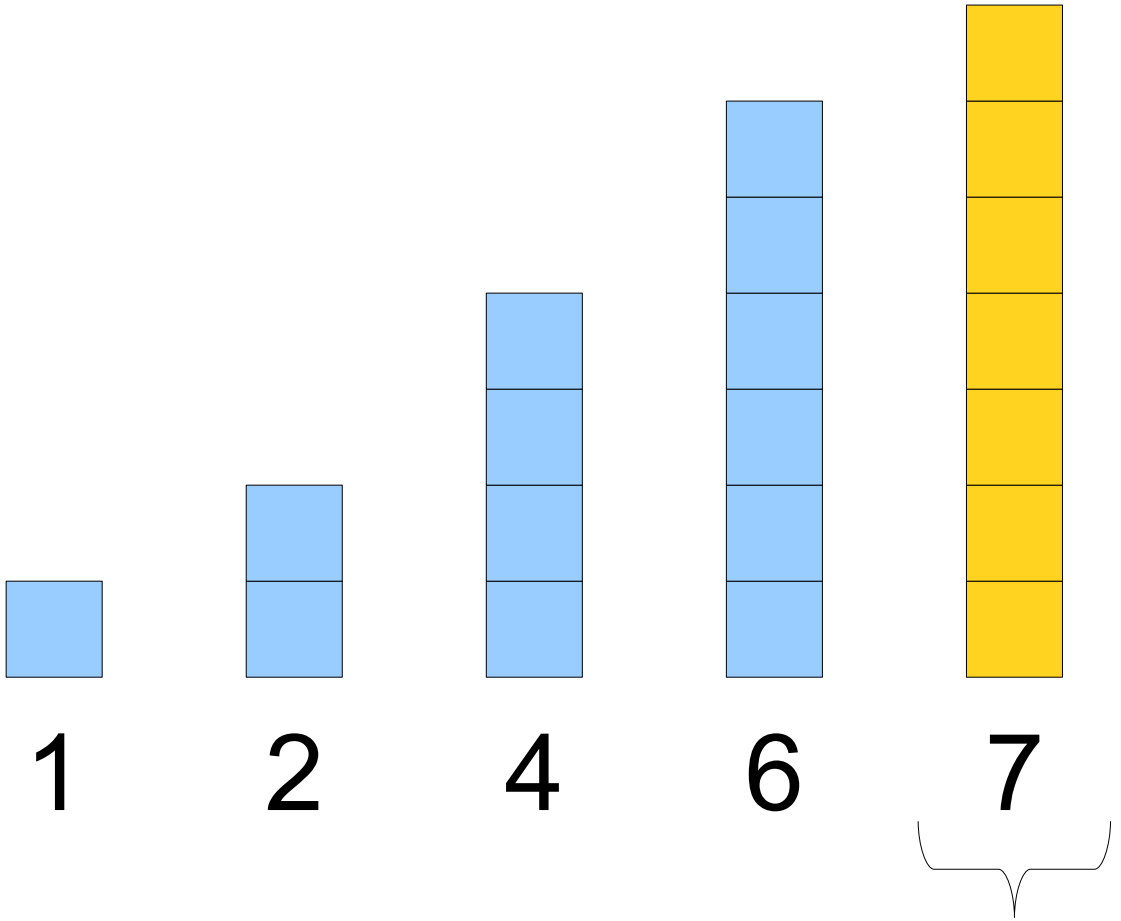
An Initial Idea: **Selection Sort**



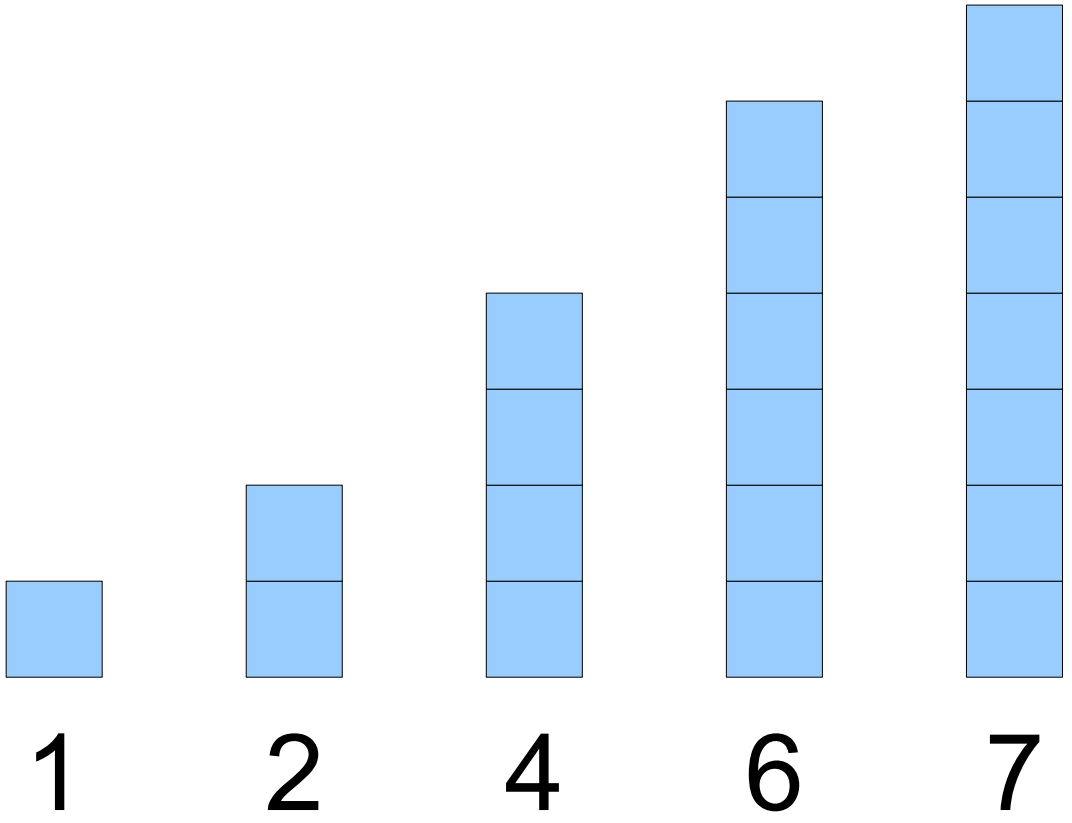
An Initial Idea: **Selection Sort**



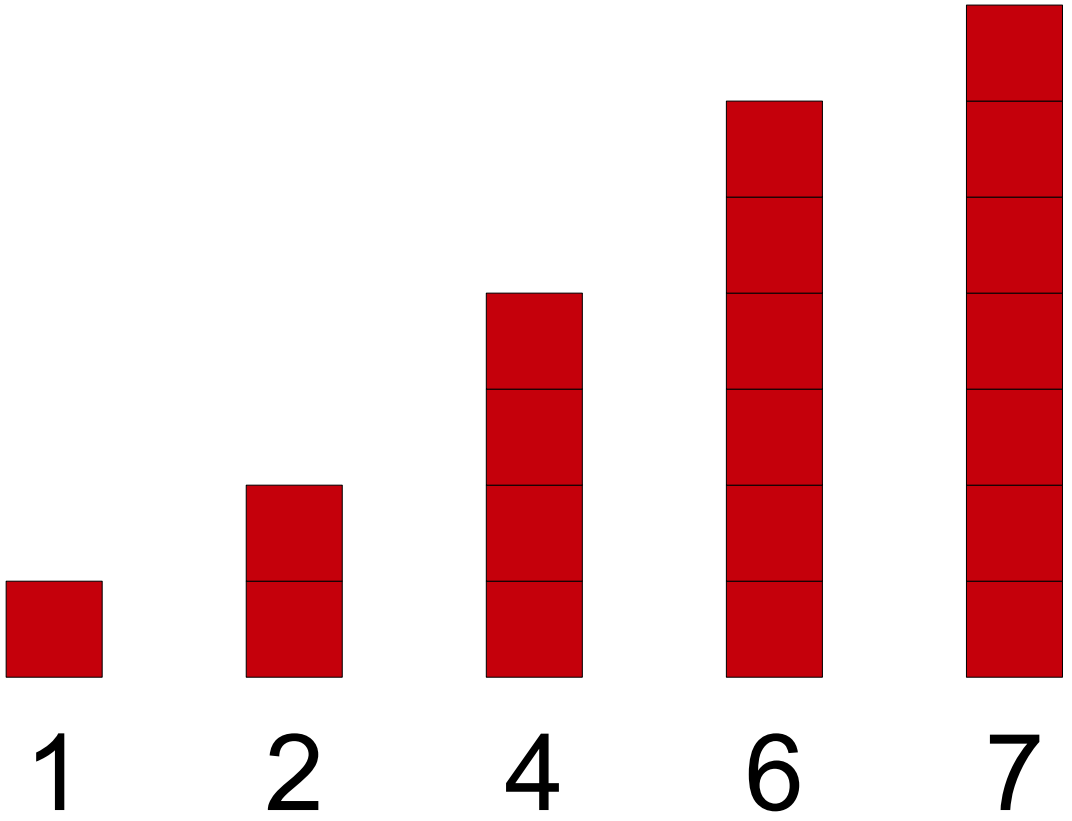
An Initial Idea: **Selection Sort**



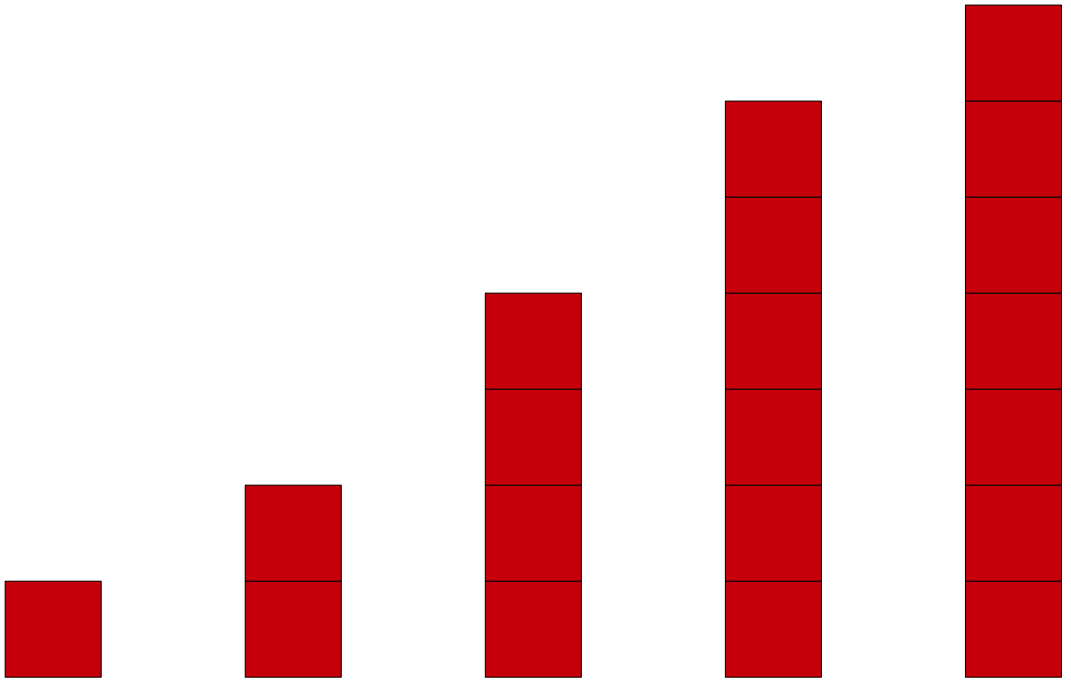
An Initial Idea: **Selection Sort**



An Initial Idea: **Selection Sort**



An Initial Idea: **Selection Sort**

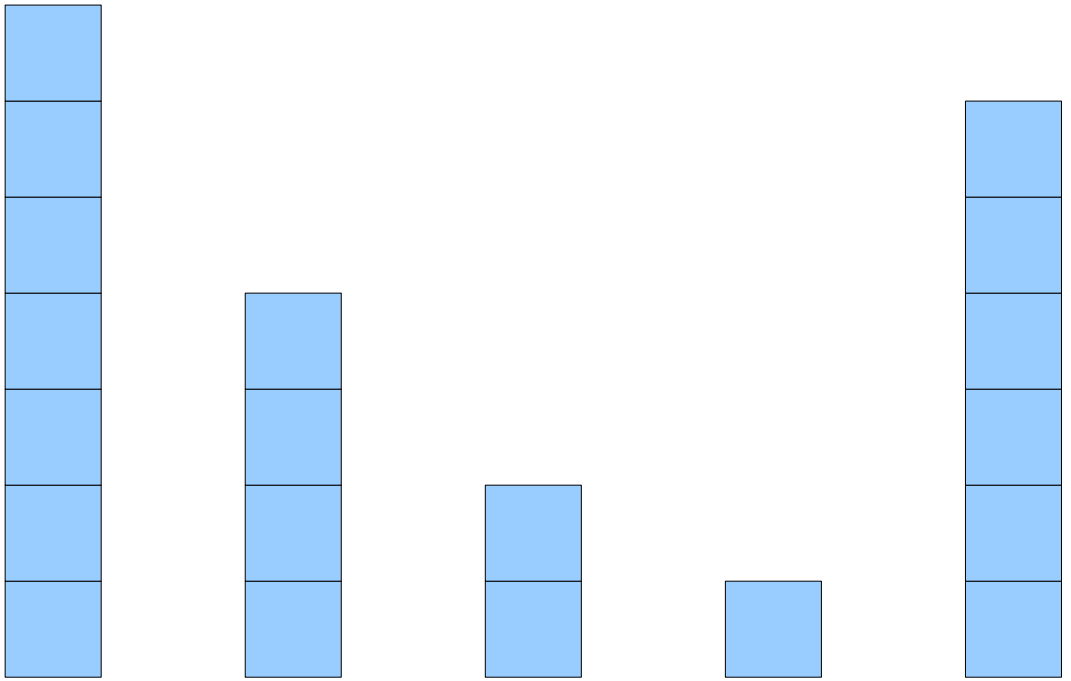


Runtime is **$O(n^2)$**

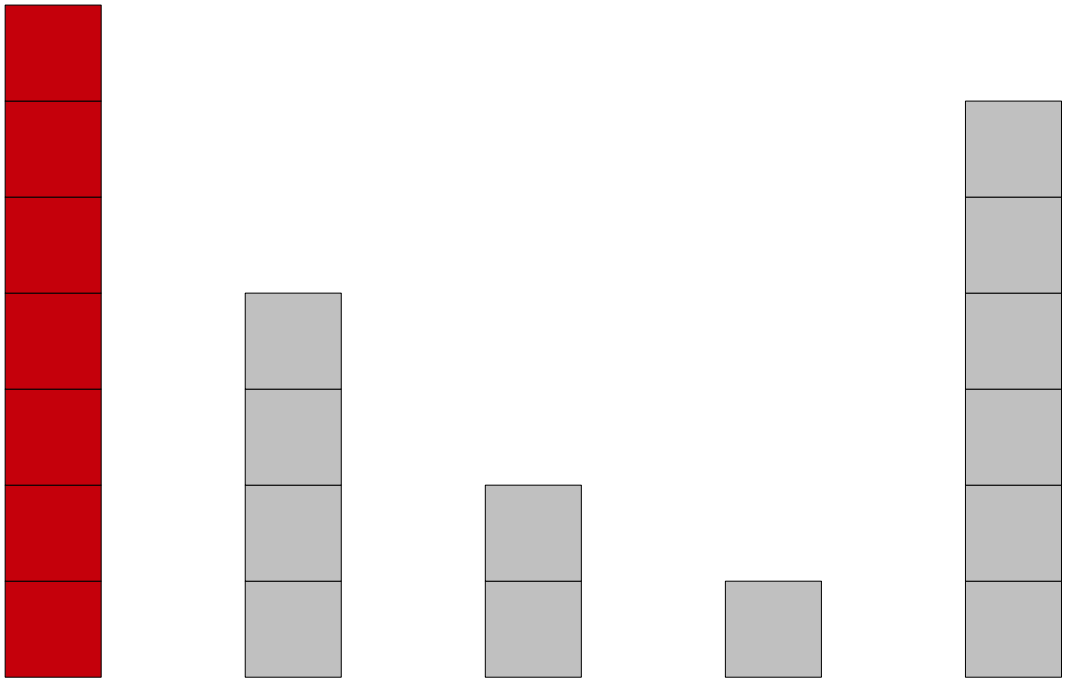
Selection Sort (Pseudocode)

Another Idea: **Insertion Sort**

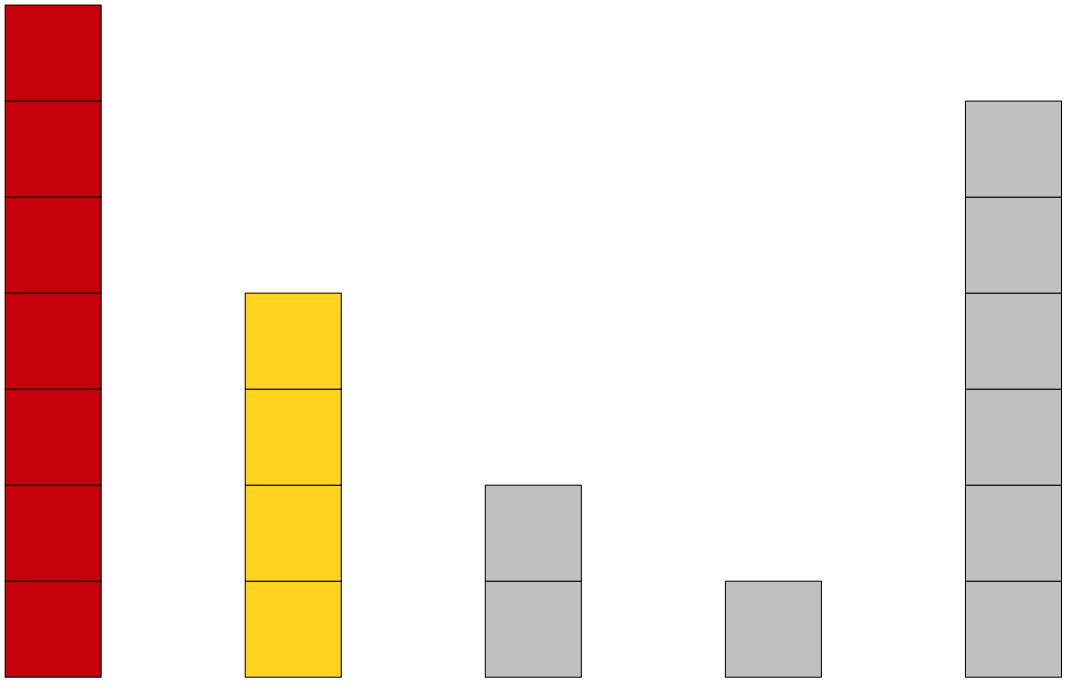
Another Idea: **Insertion Sort**



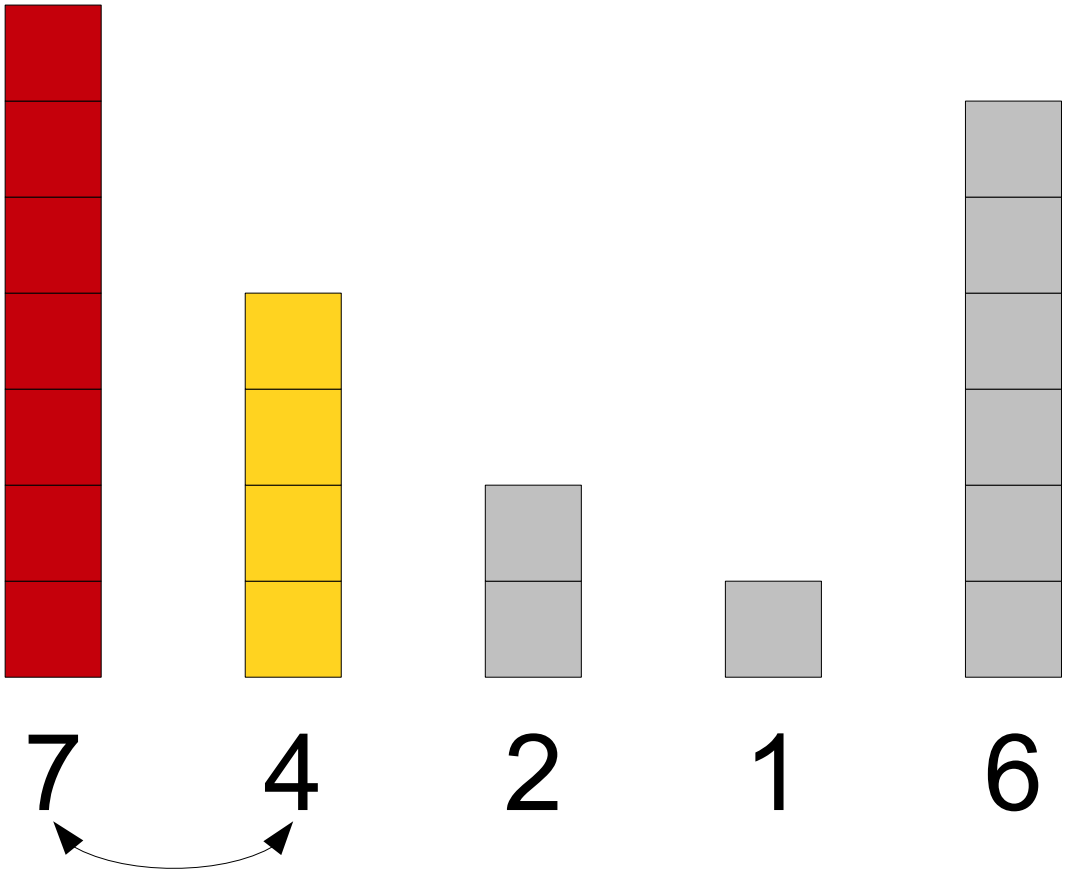
Another Idea: **Insertion Sort**



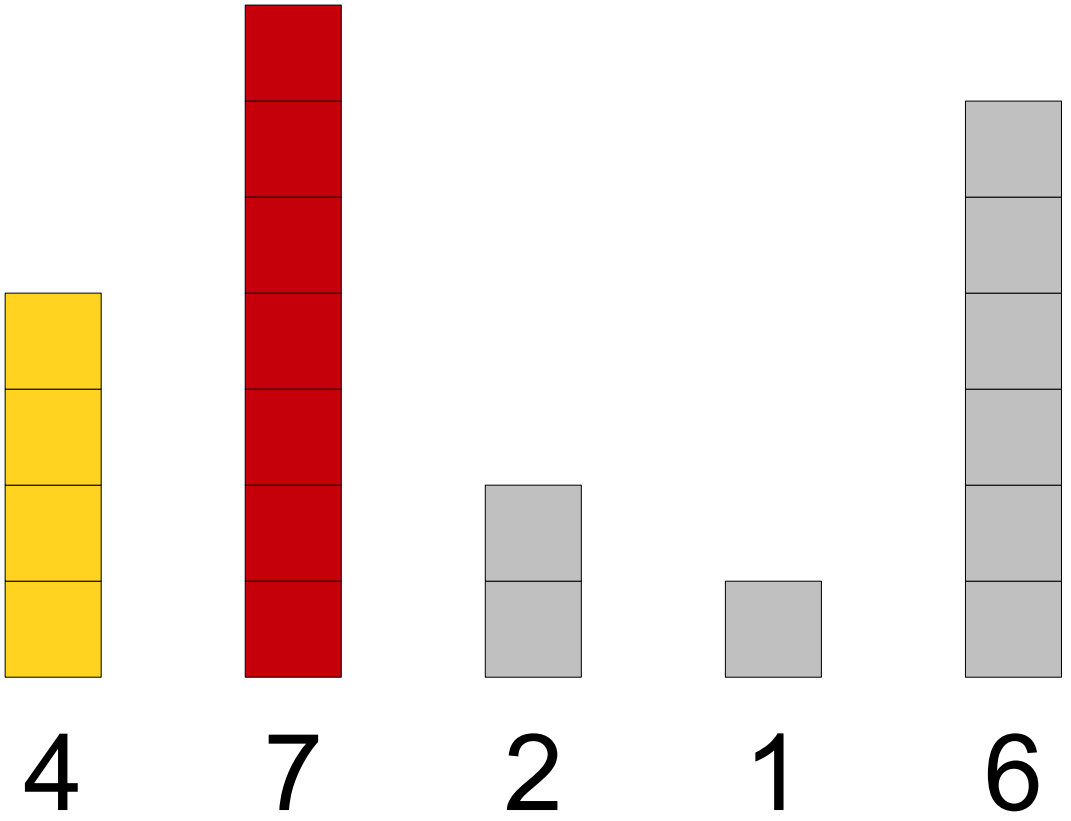
Another Idea: **Insertion Sort**



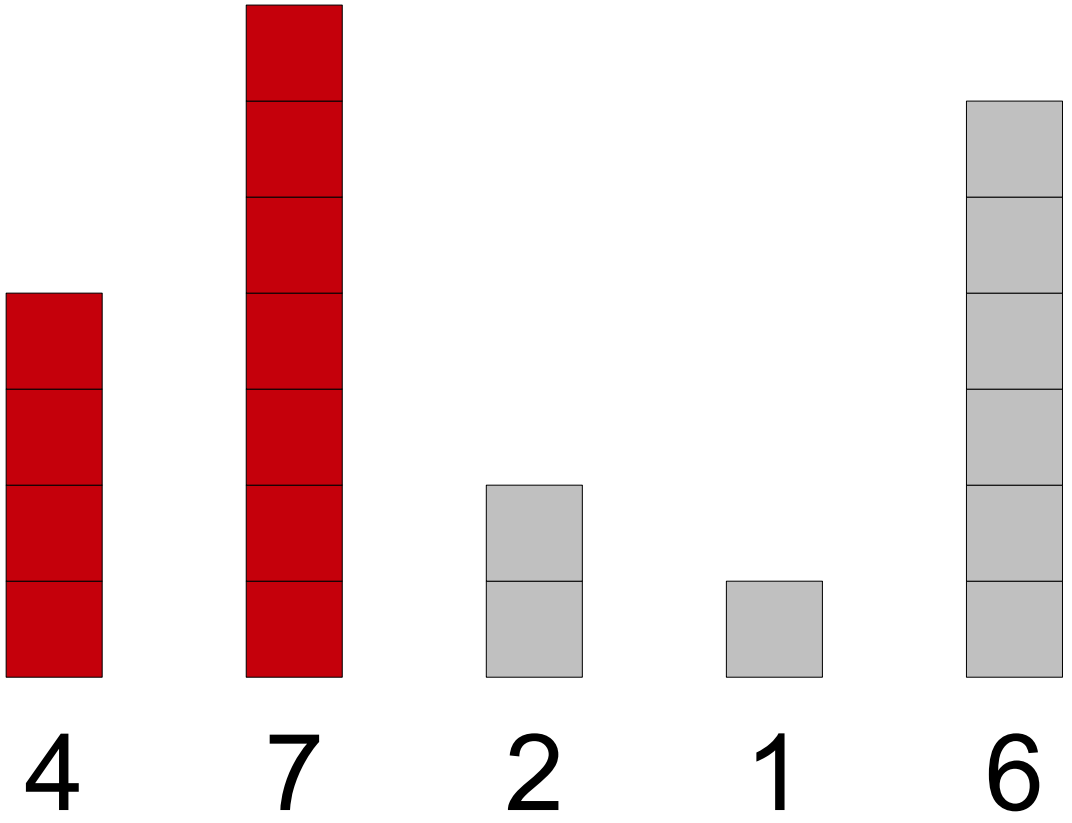
Another Idea: **Insertion Sort**



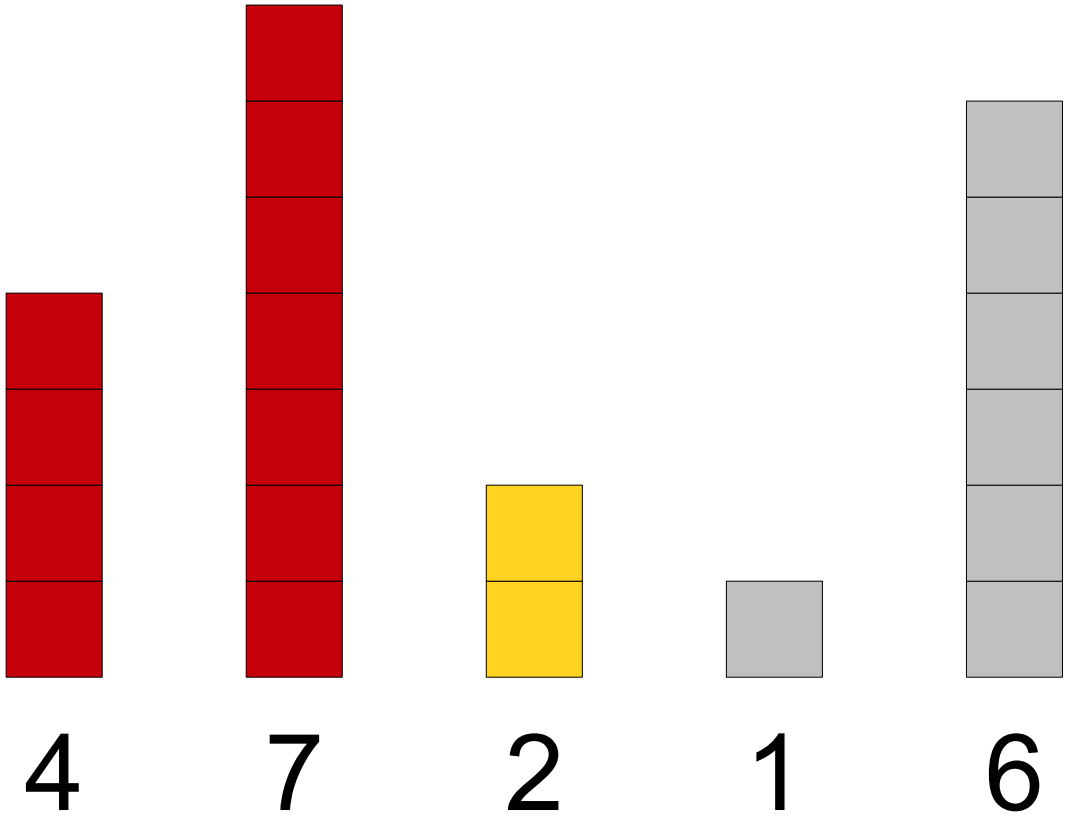
Another Idea: **Insertion Sort**



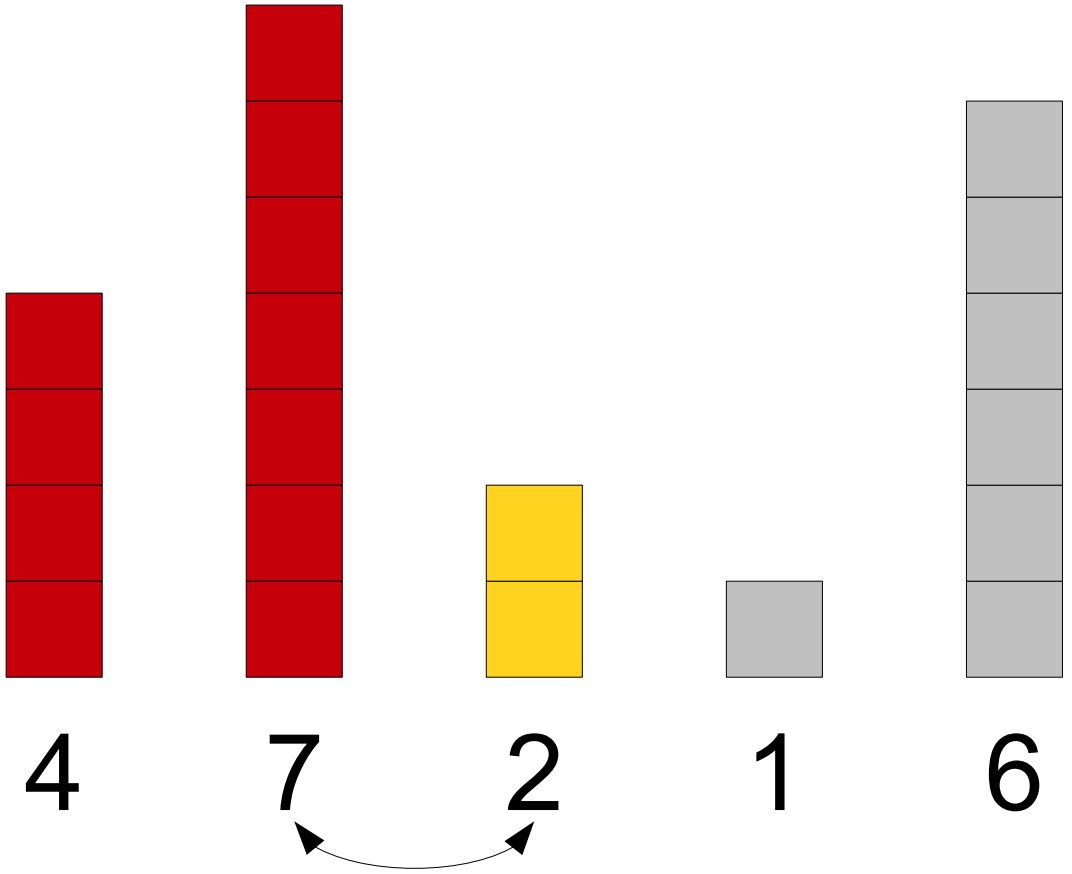
Another Idea: **Insertion Sort**



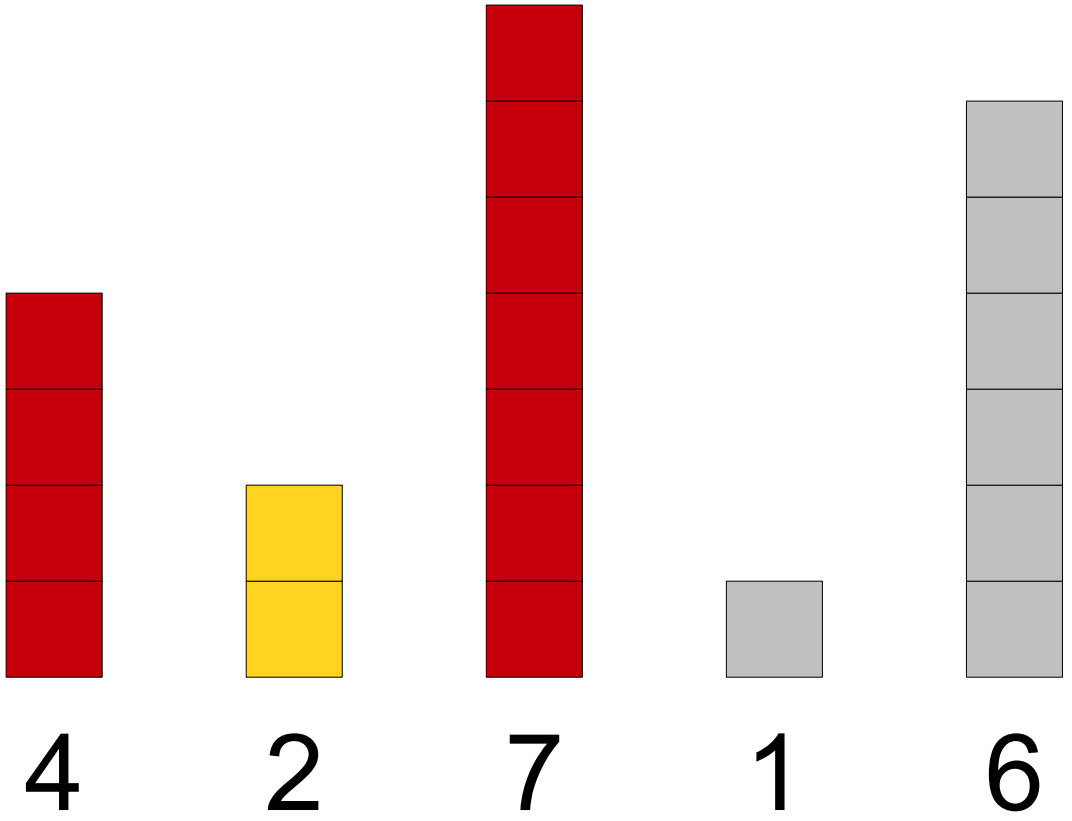
Another Idea: **Insertion Sort**



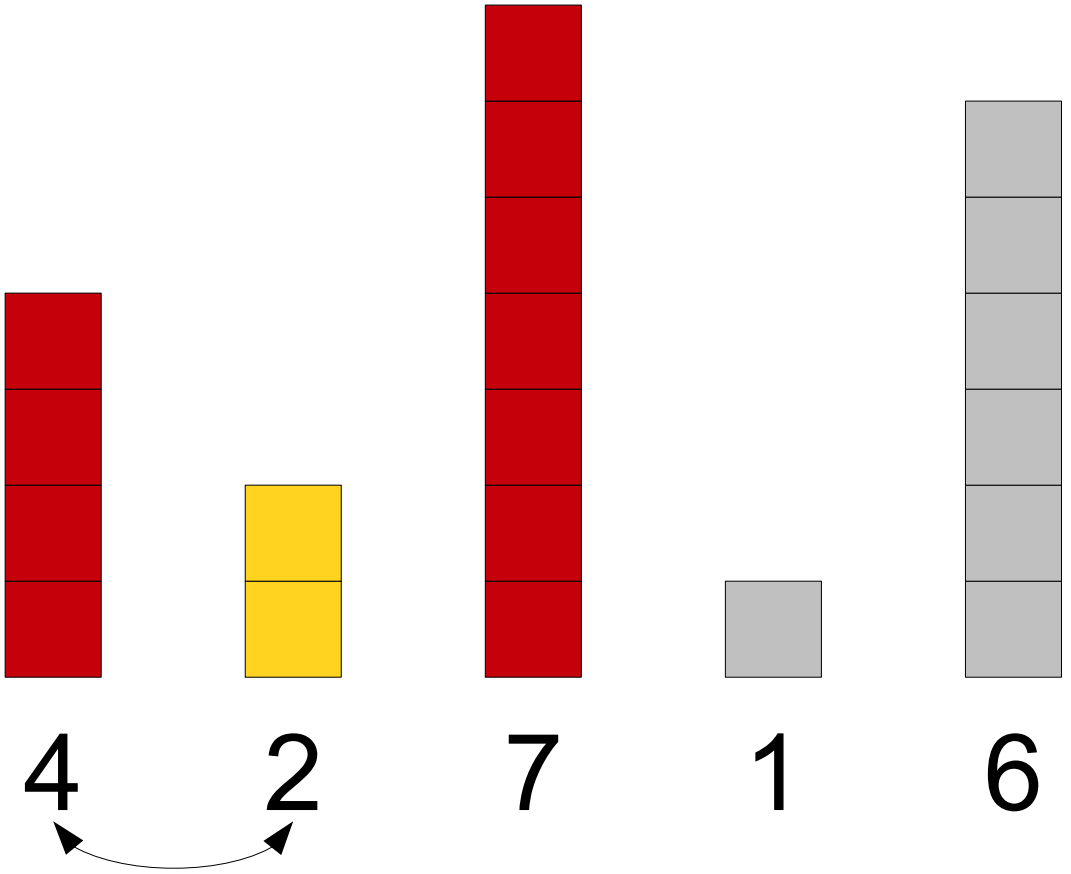
Another Idea: **Insertion Sort**



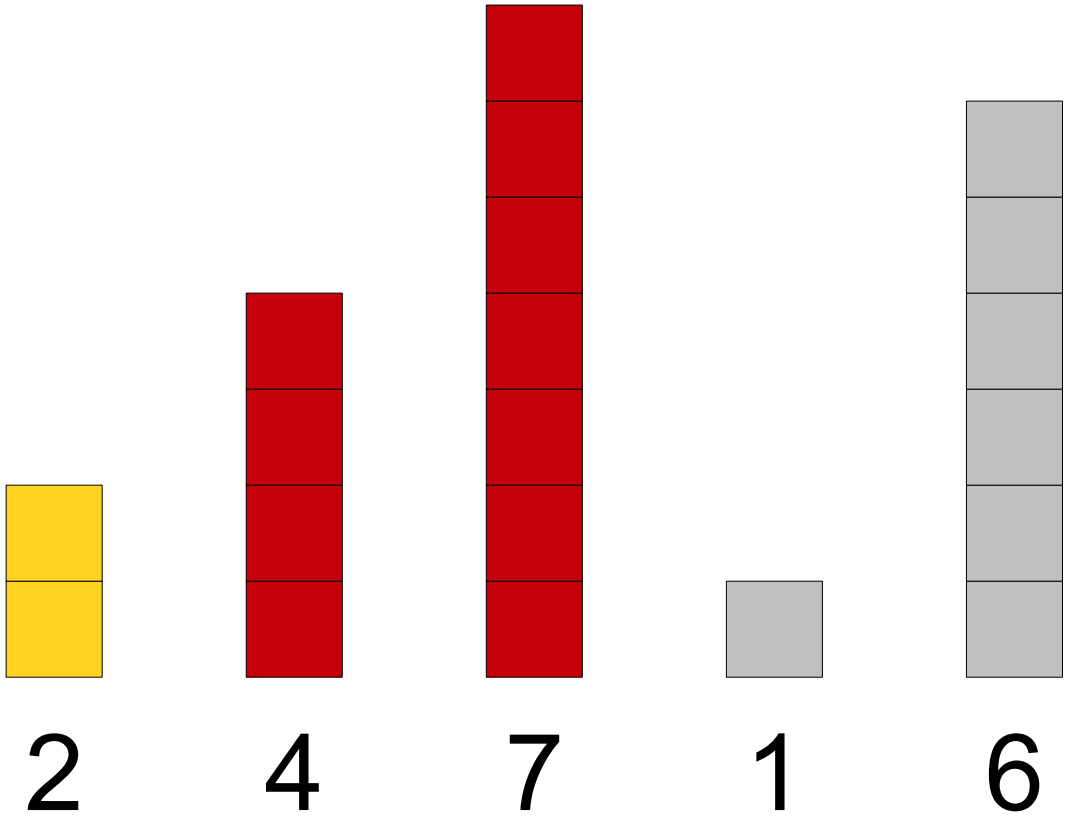
Another Idea: **Insertion Sort**



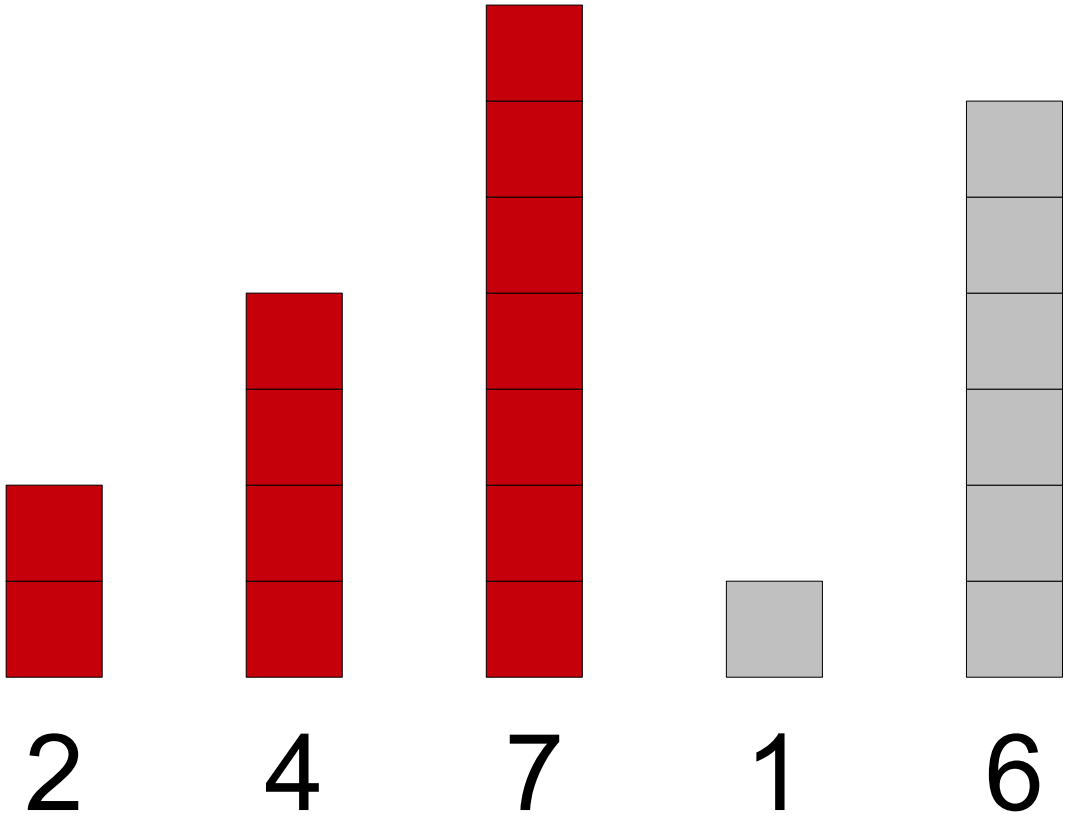
Another Idea: **Insertion Sort**



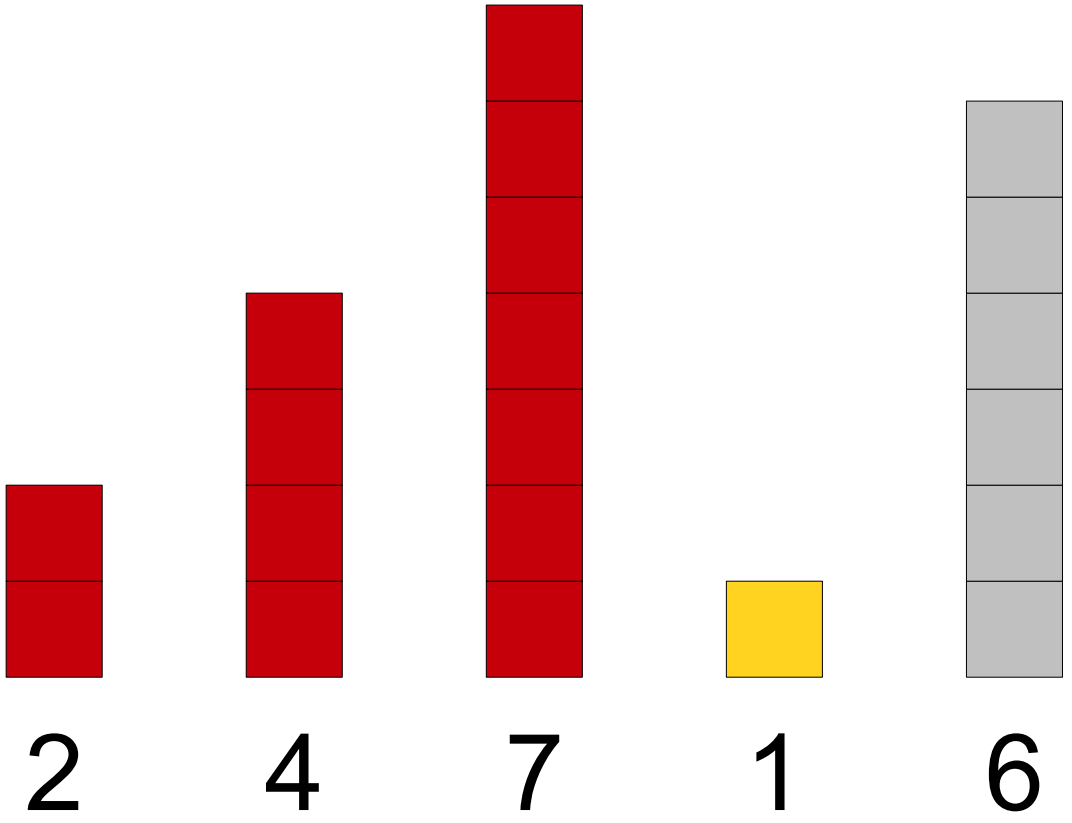
Another Idea: **Insertion Sort**



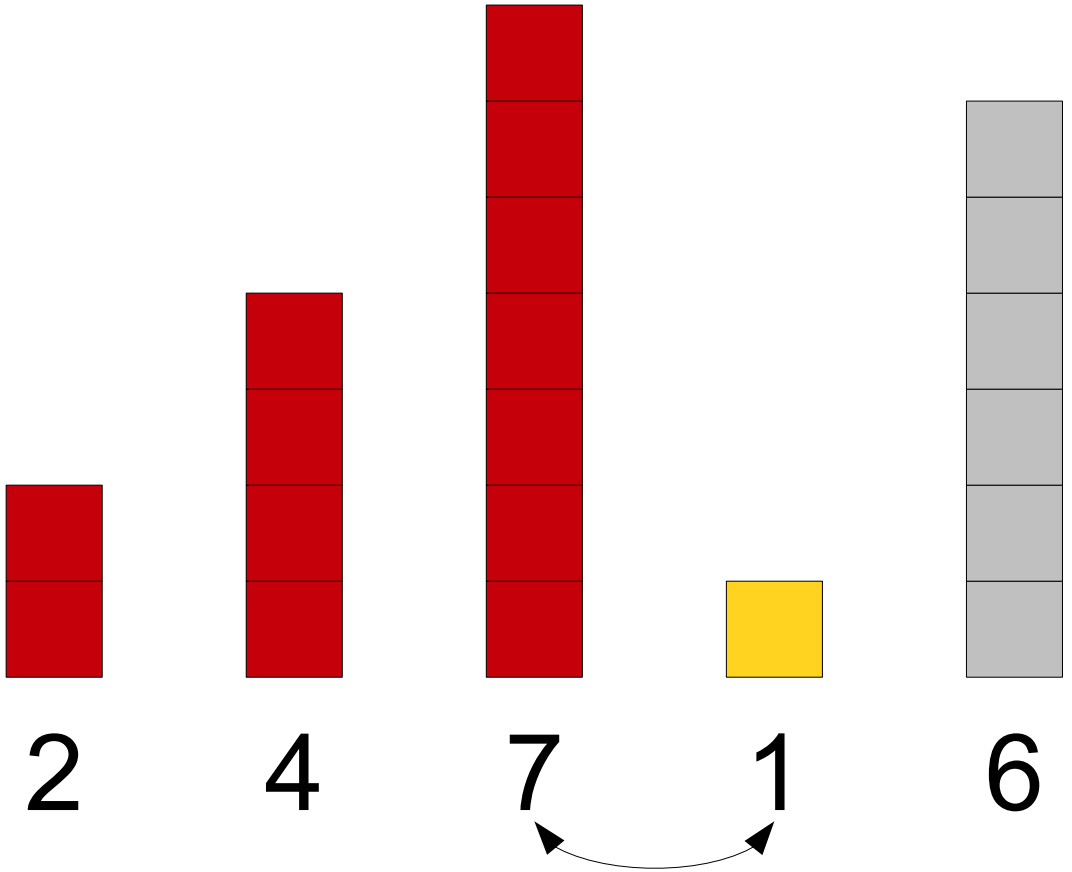
Another Idea: **Insertion Sort**



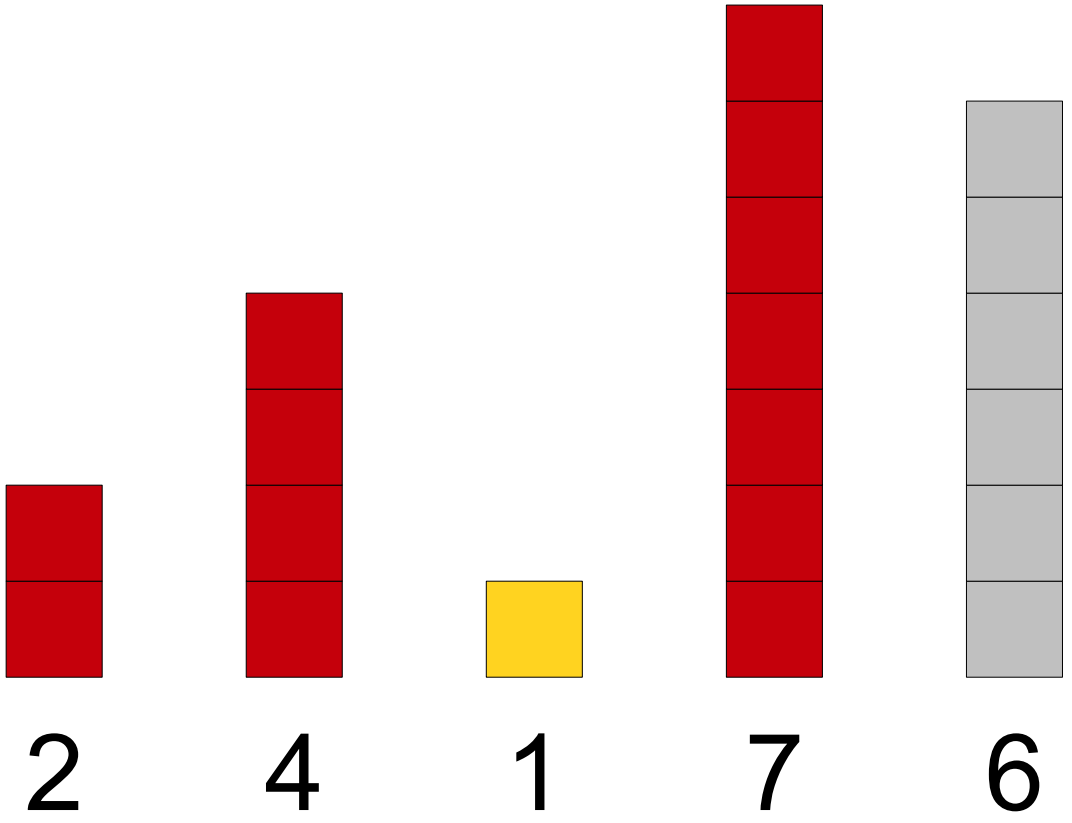
Another Idea: **Insertion Sort**



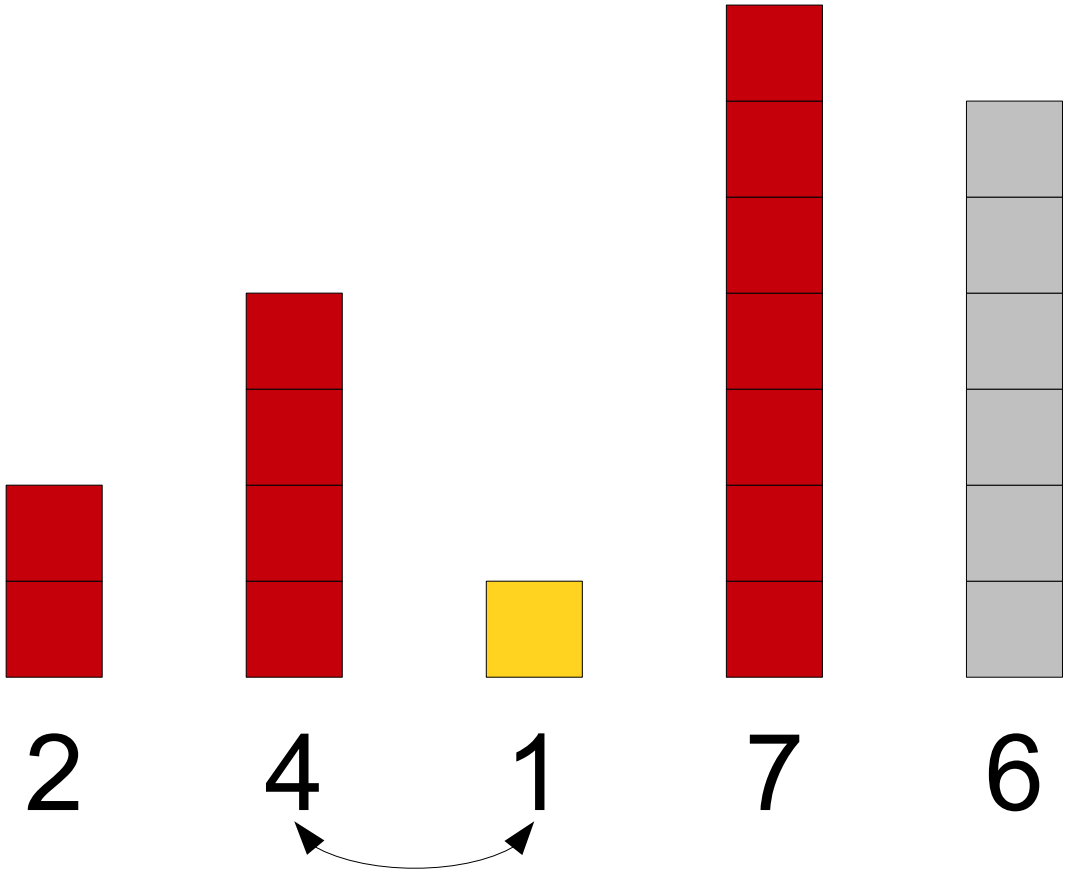
Another Idea: **Insertion Sort**



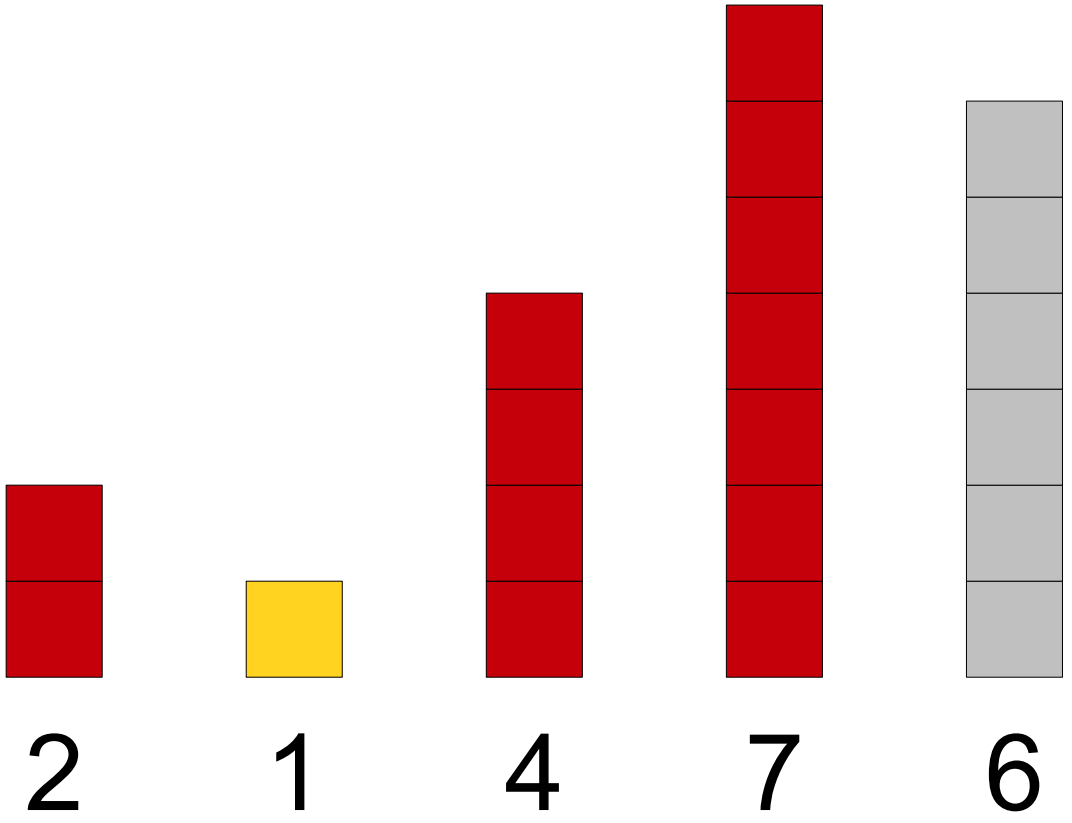
Another Idea: **Insertion Sort**



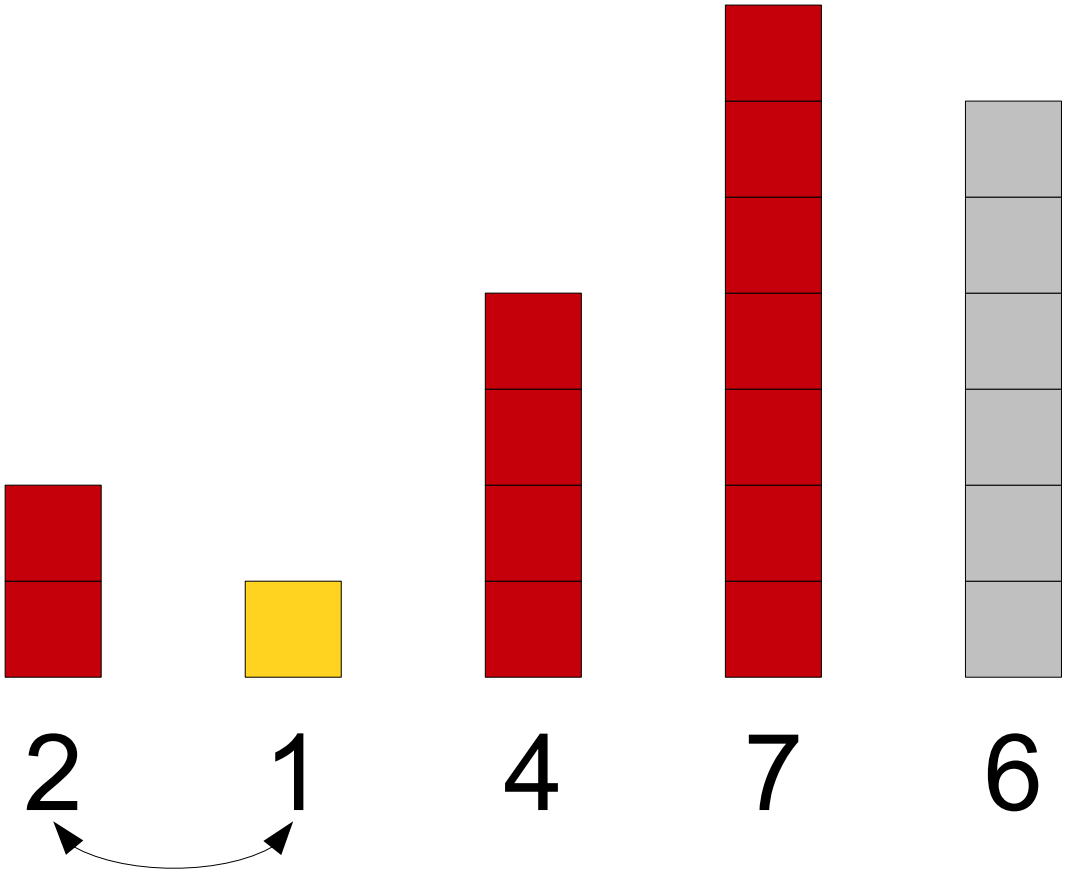
Another Idea: **Insertion Sort**



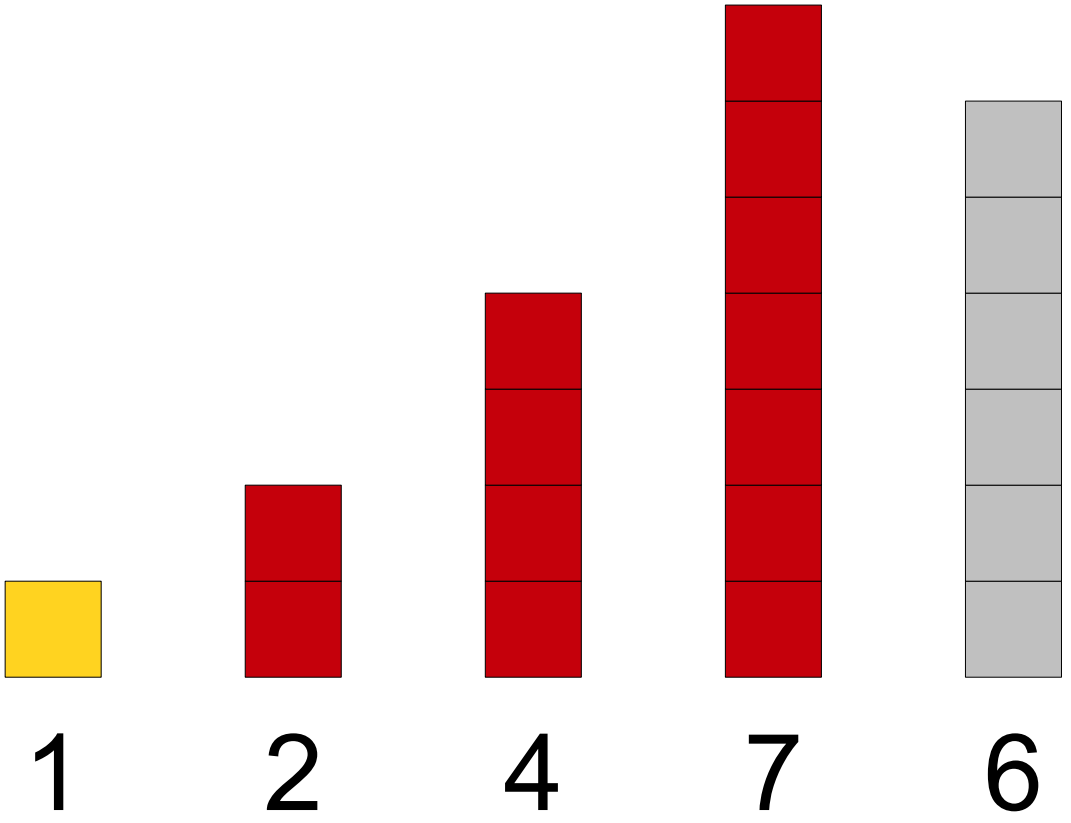
Another Idea: **Insertion Sort**



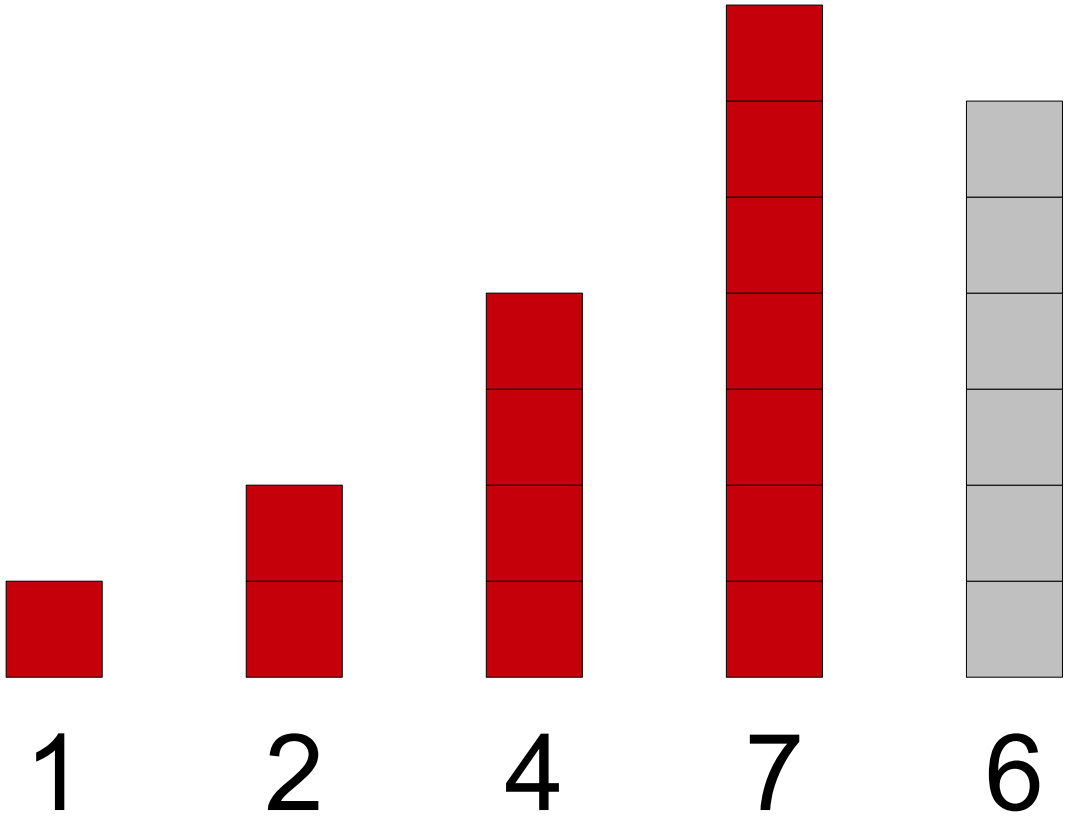
Another Idea: **Insertion Sort**



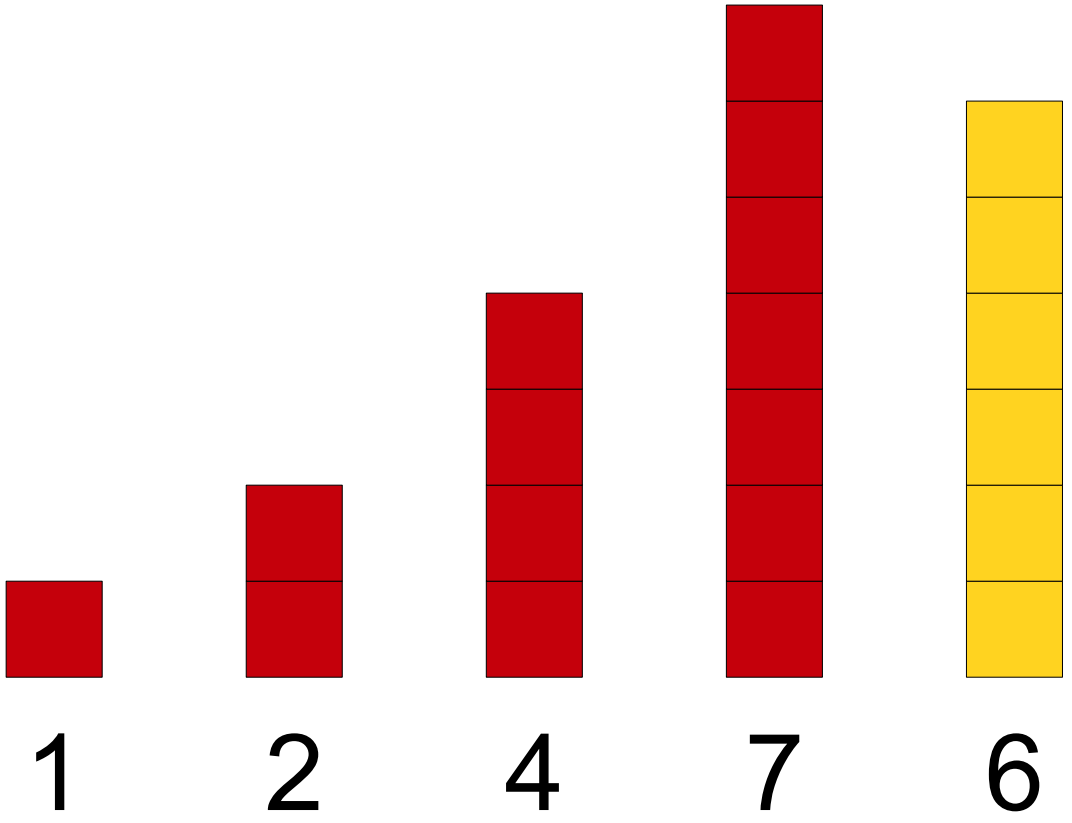
Another Idea: **Insertion Sort**



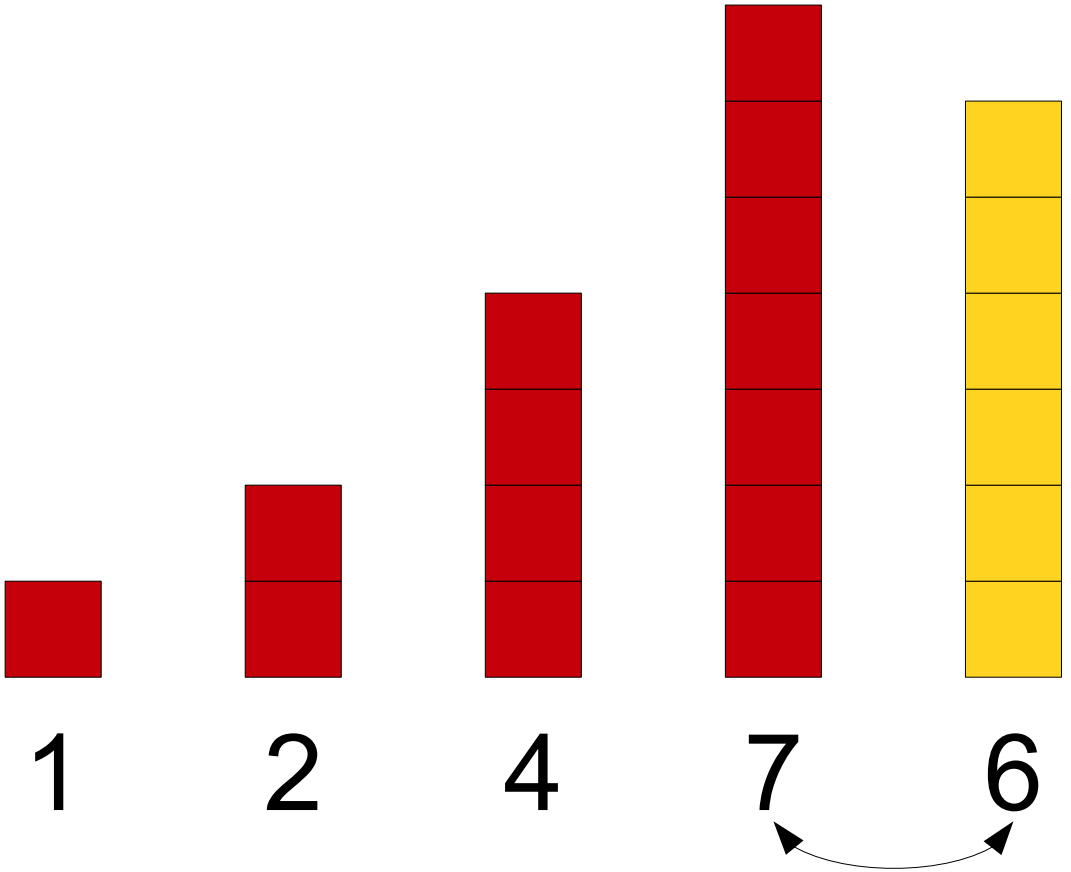
Another Idea: **Insertion Sort**



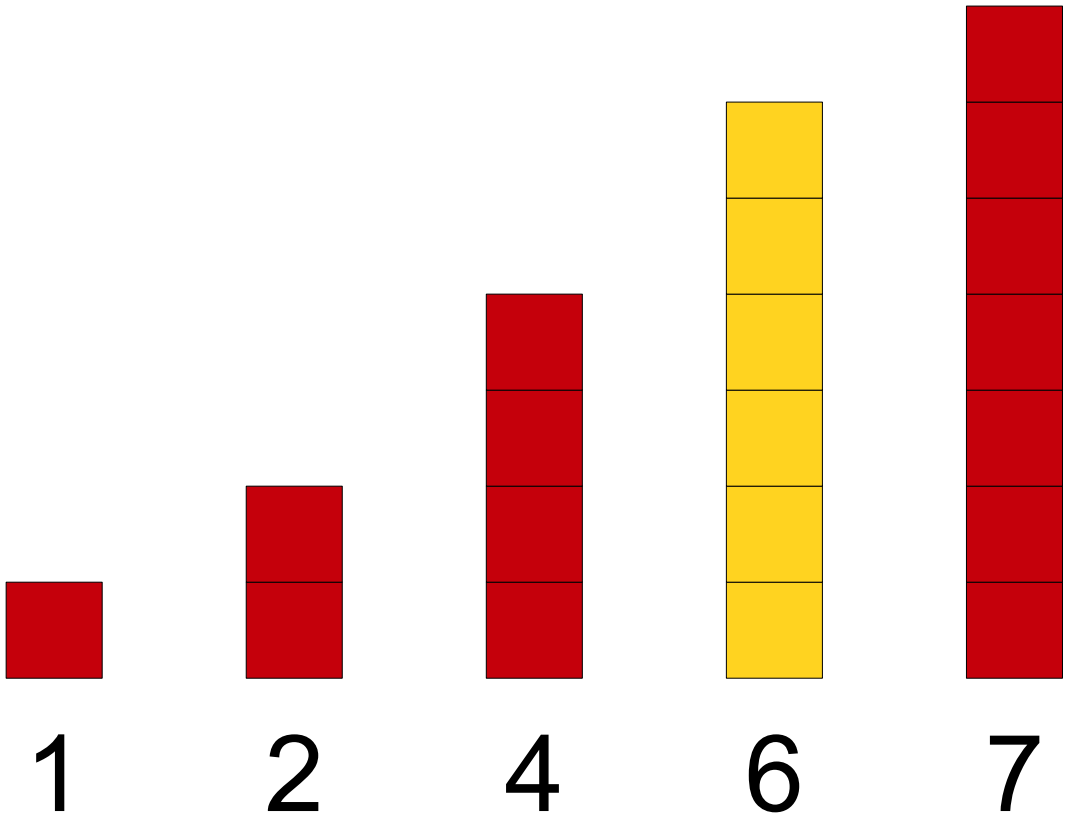
Another Idea: **Insertion Sort**



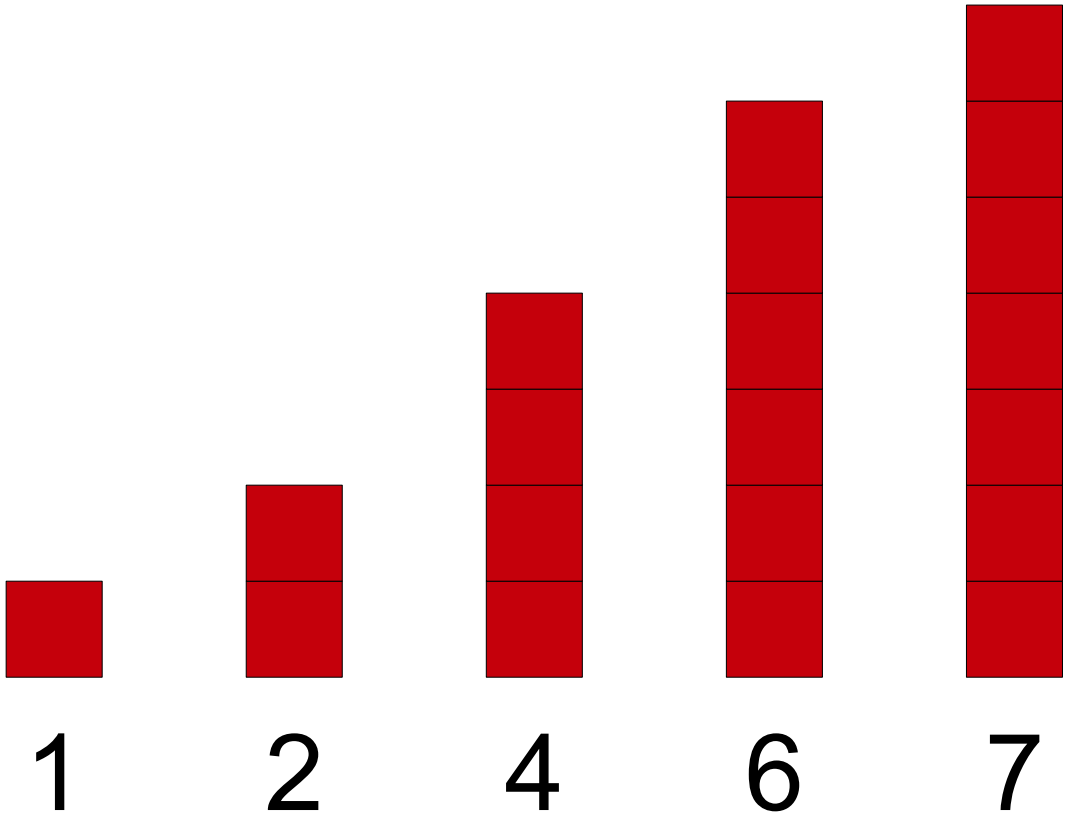
Another Idea: **Insertion Sort**



Another Idea: **Insertion Sort**



Another Idea: **Insertion Sort**

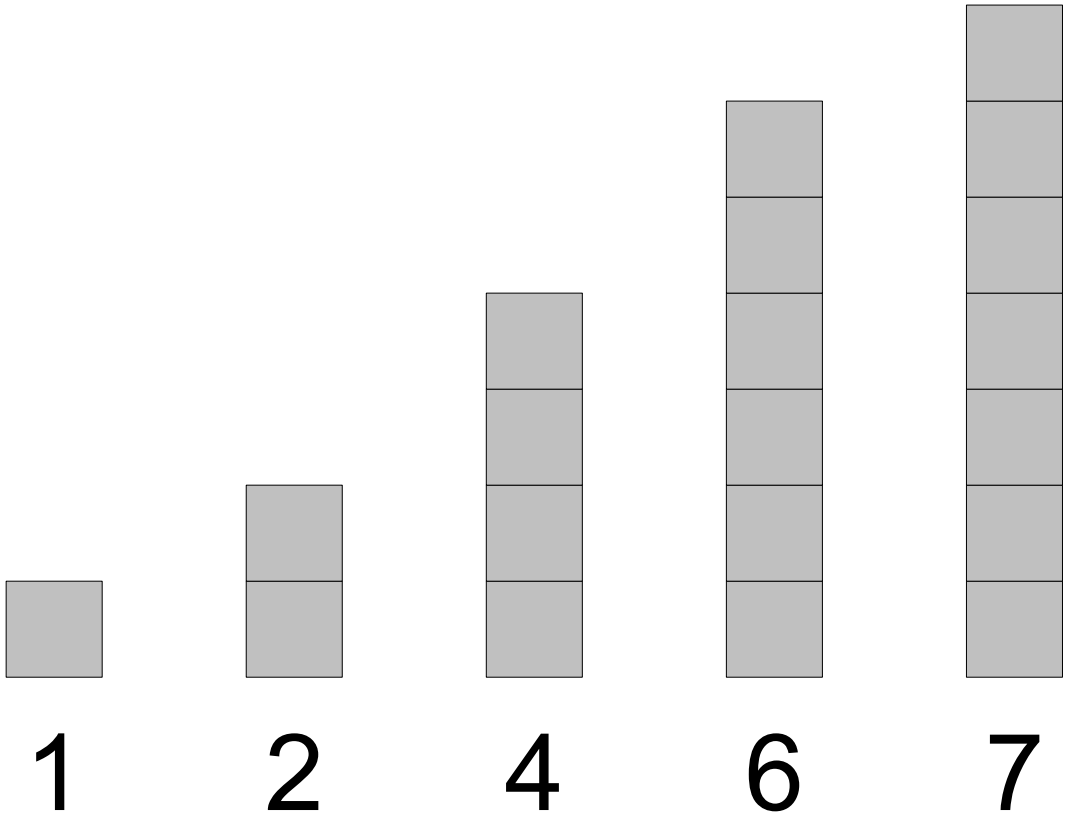


Insertion Sort (Pseudocode)

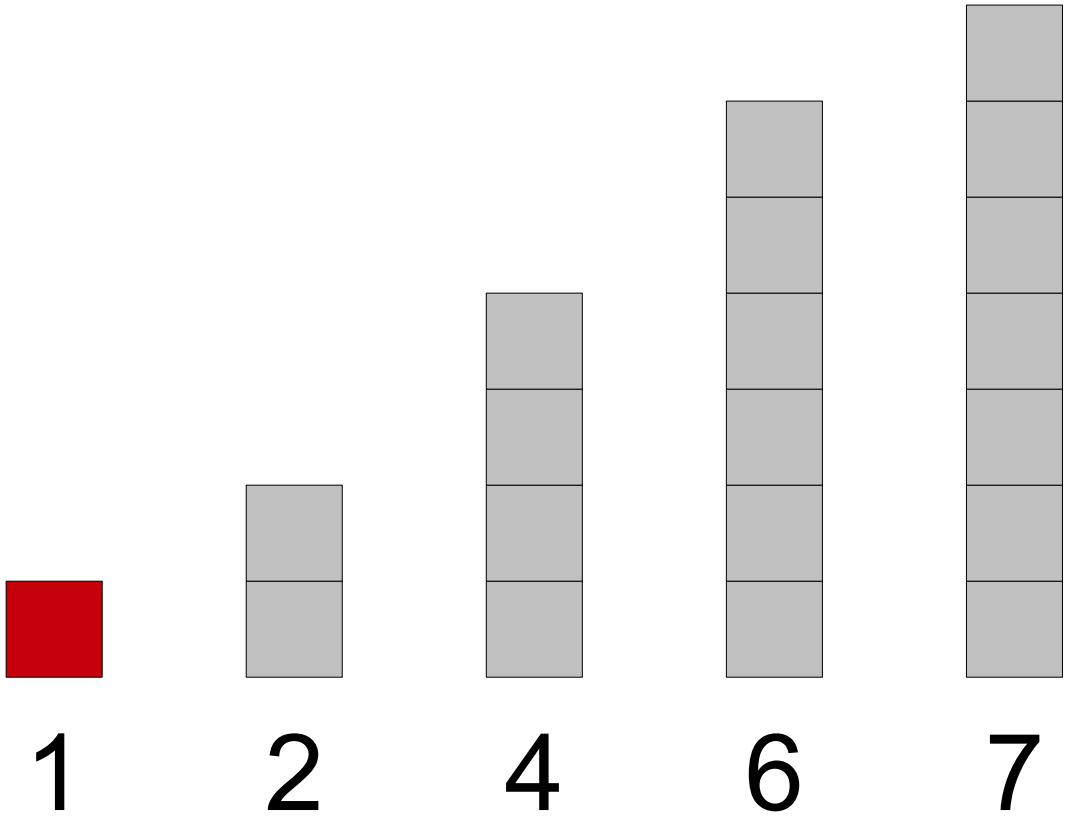
Insertion Sort in Code

```
void insertionSort(Vector<int>& v) {
 for (int i = 0; i < v.size(); i++) {
   for (int j = i - 1; j >= 0; j--) {
     if (v[j] < v[j + 1]) break;
     swap(v[j], v[j + 1]);
   }
  }
}
```

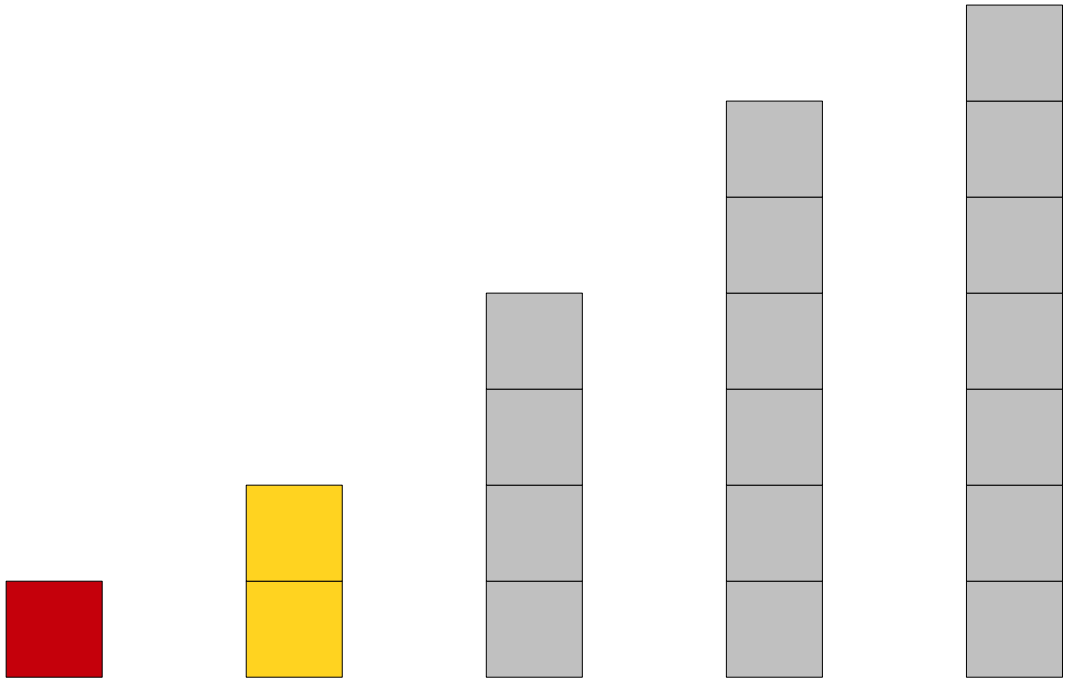
How Fast is Insertion Sort?



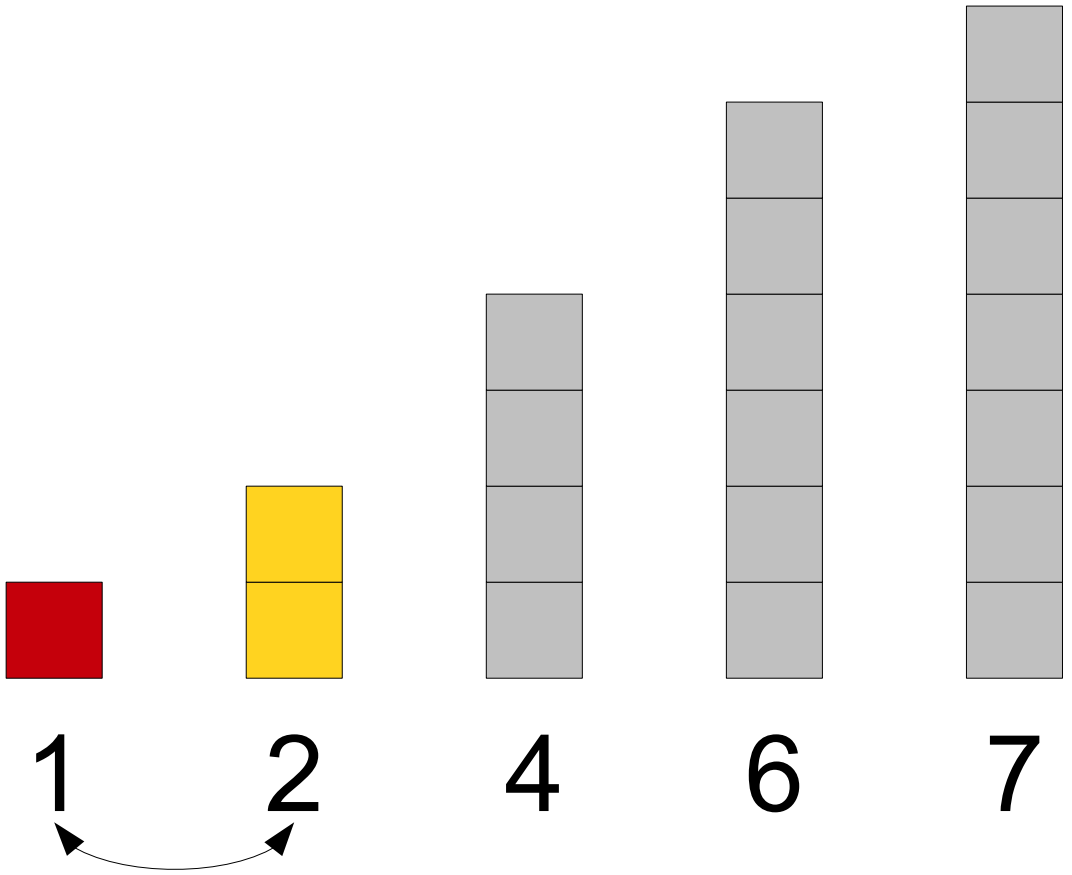
How Fast is Insertion Sort?



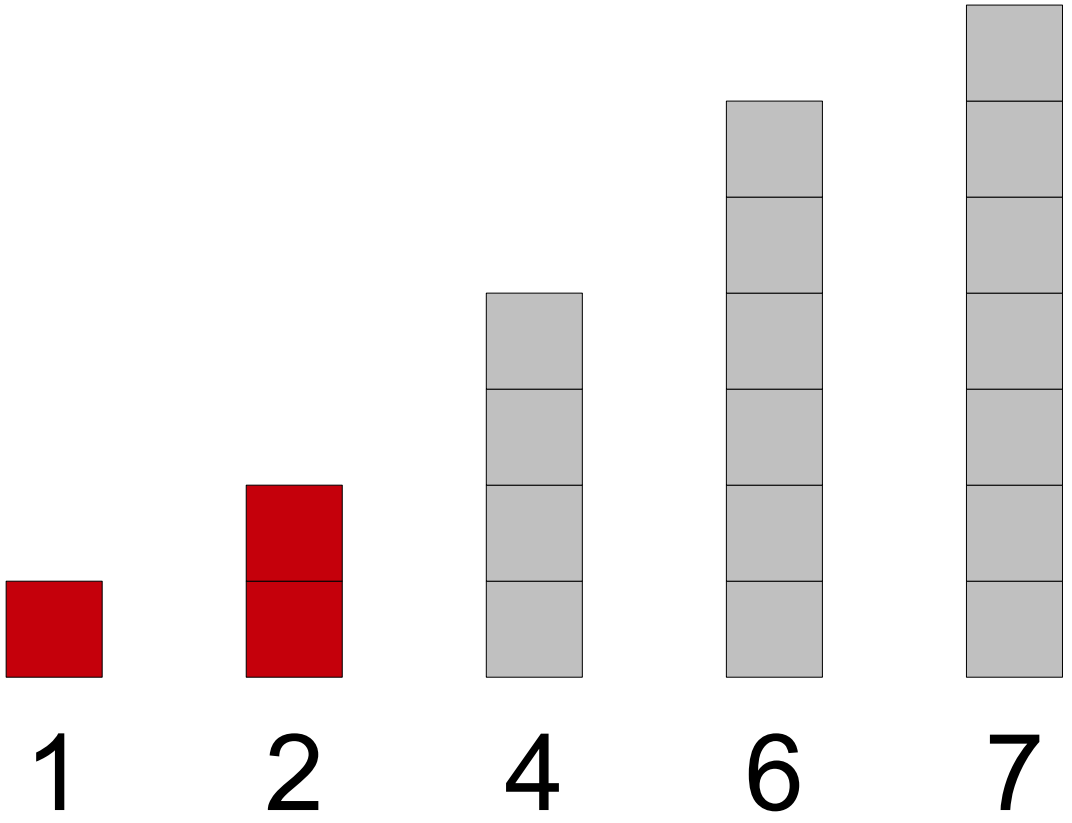
How Fast is Insertion Sort?



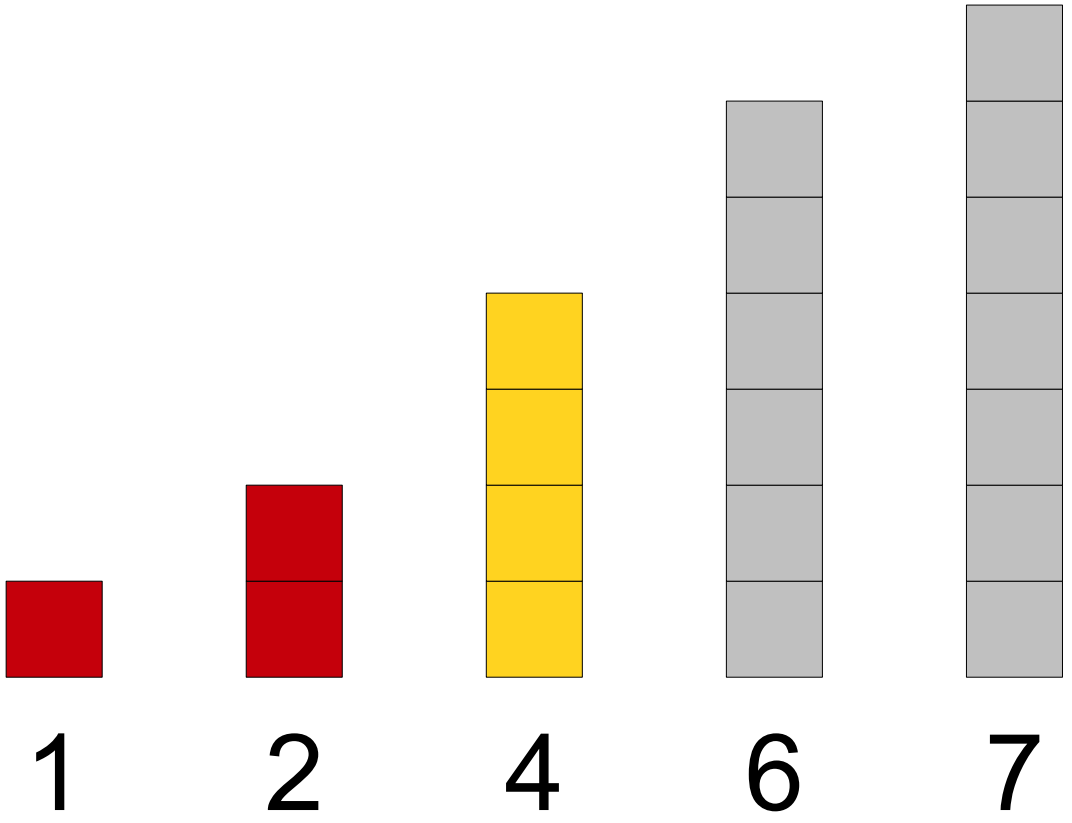
How Fast is Insertion Sort?



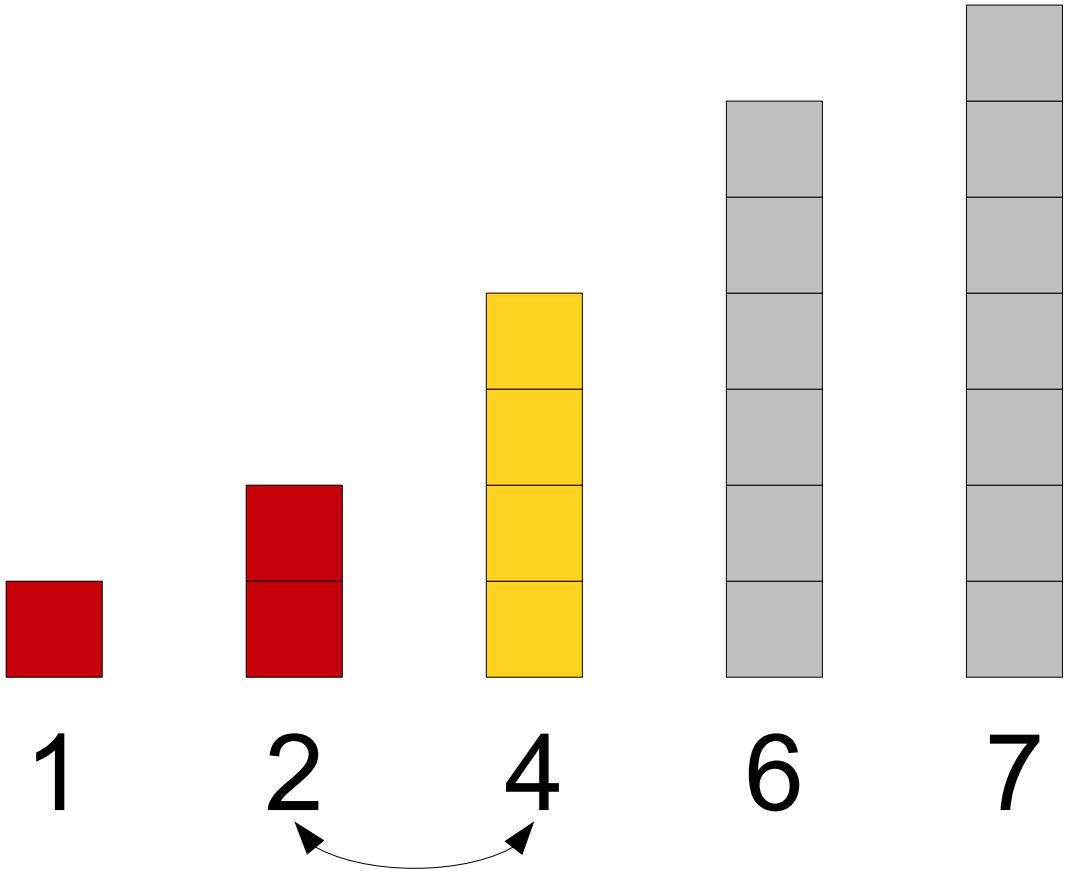
How Fast is Insertion Sort?



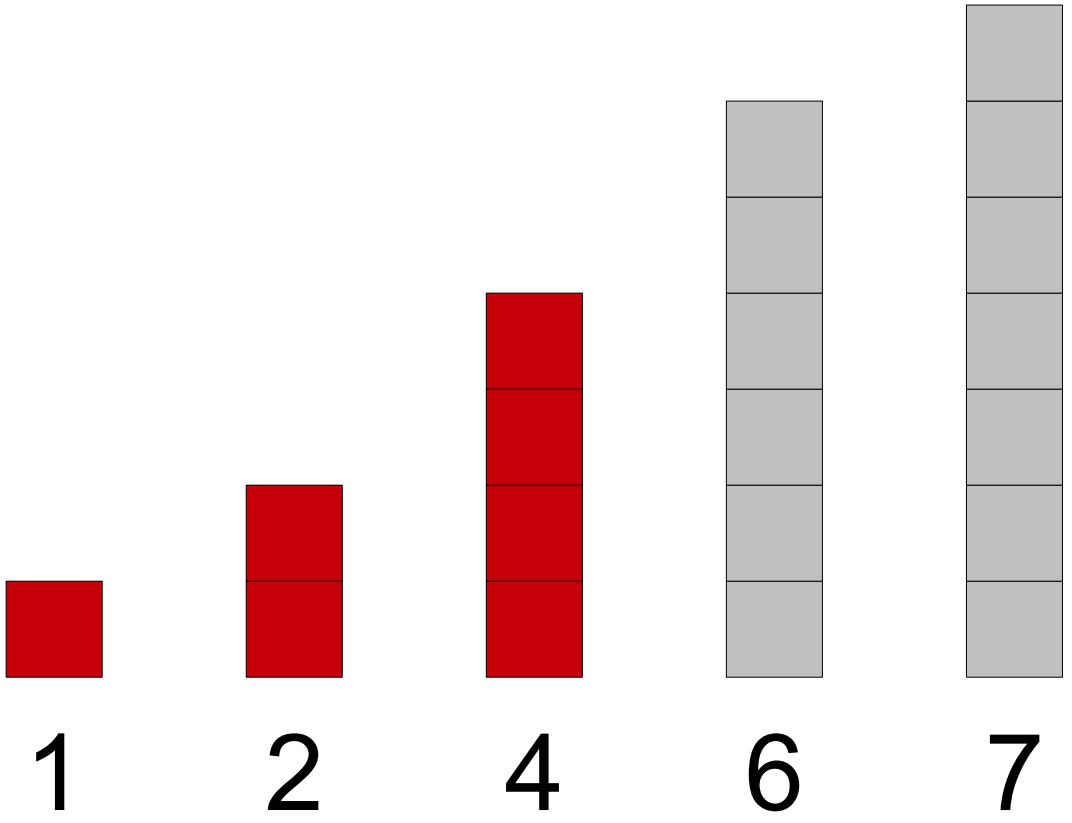
How Fast is Insertion Sort?



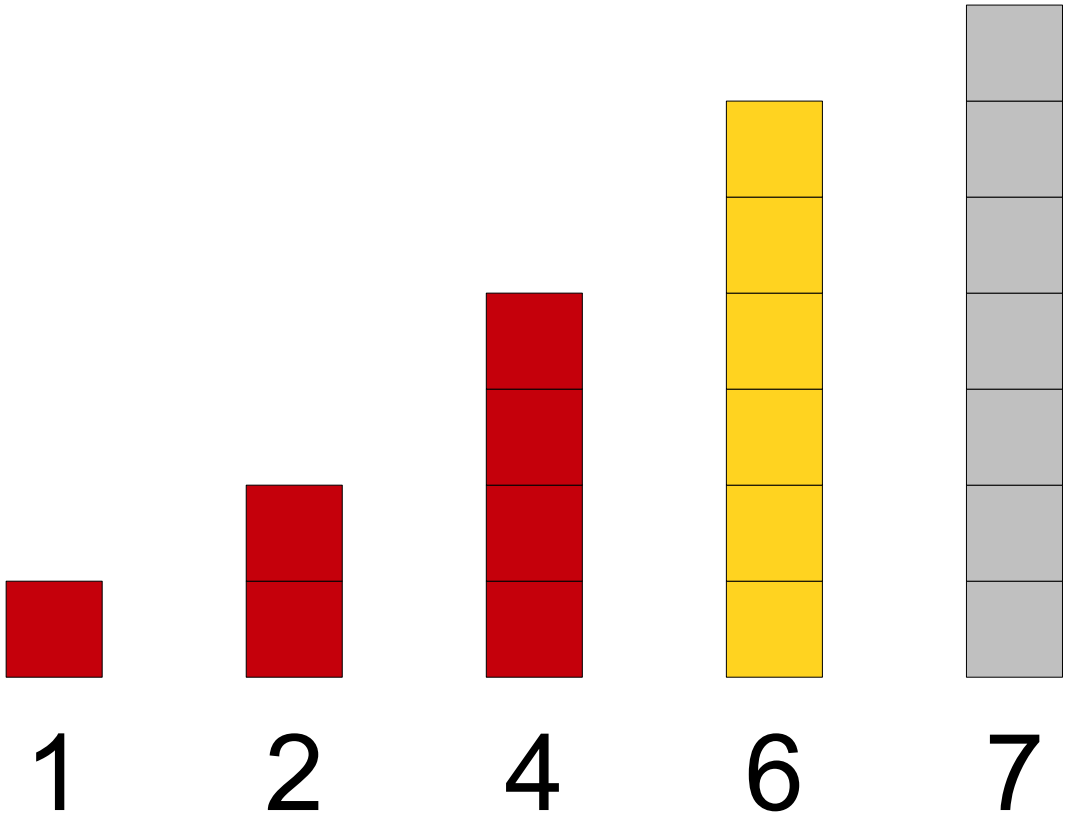
How Fast is Insertion Sort?



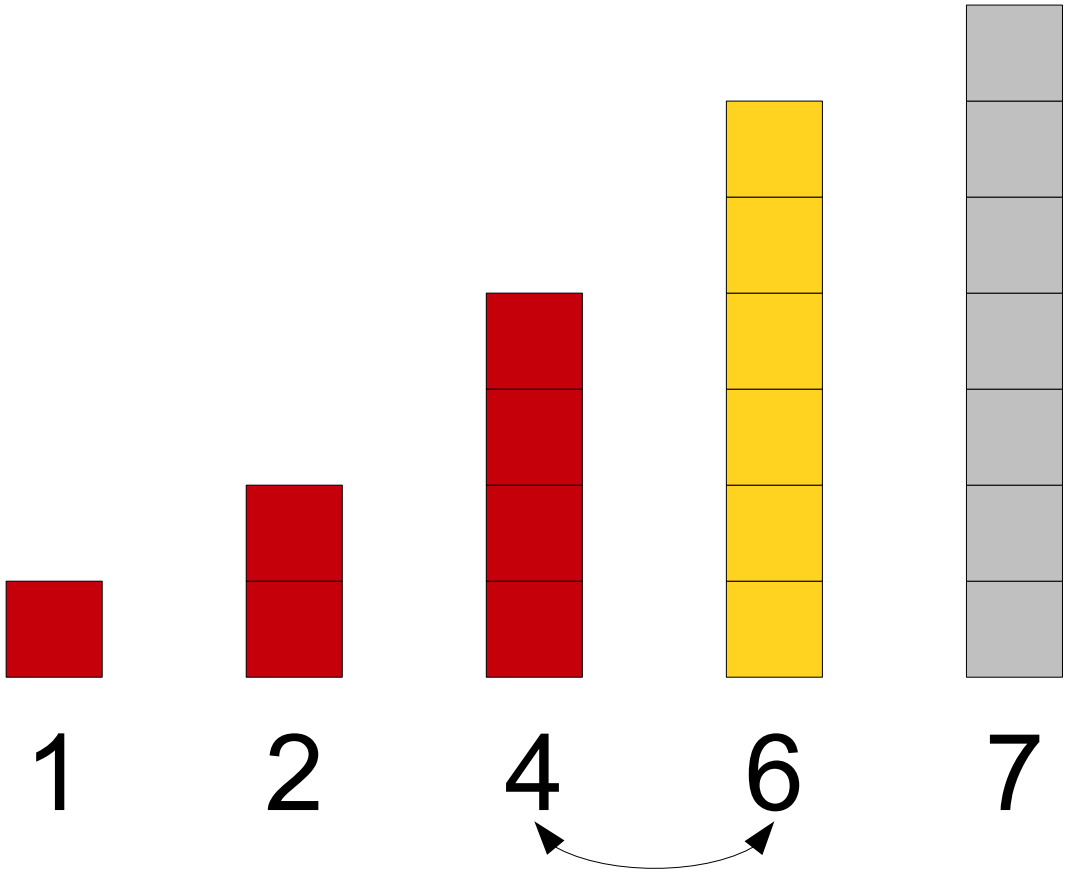
How Fast is Insertion Sort?



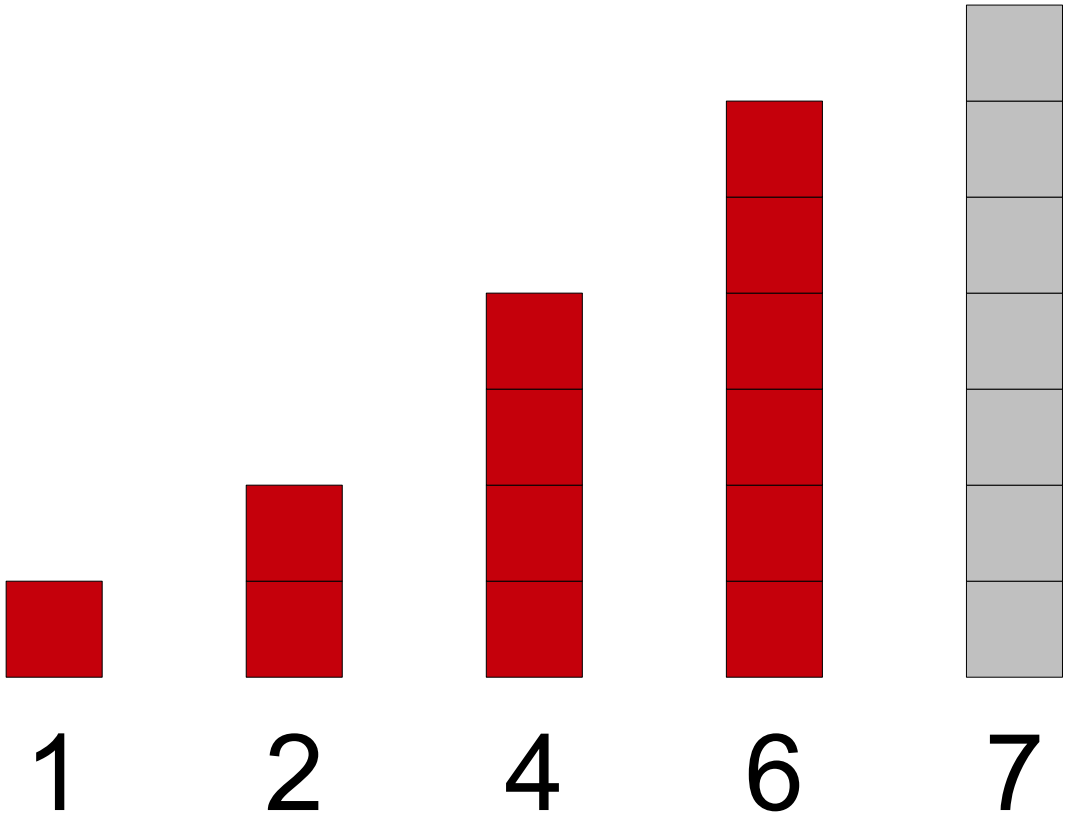
How Fast is Insertion Sort?



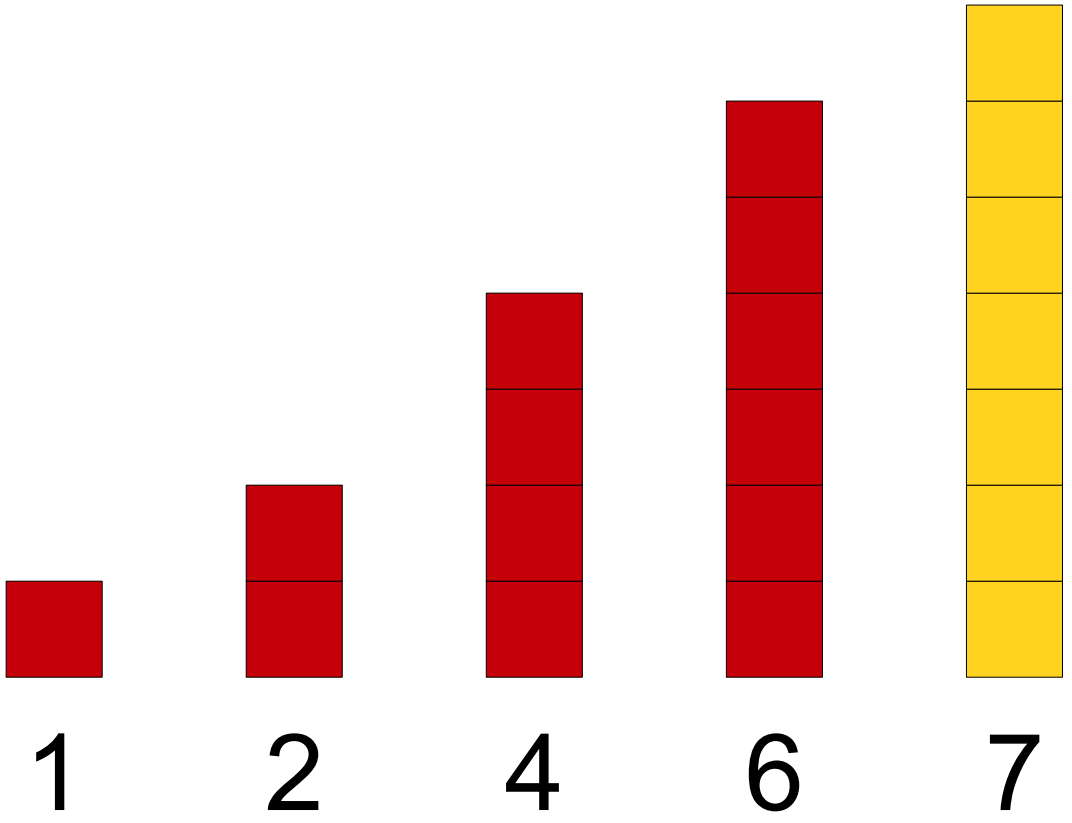
How Fast is Insertion Sort?



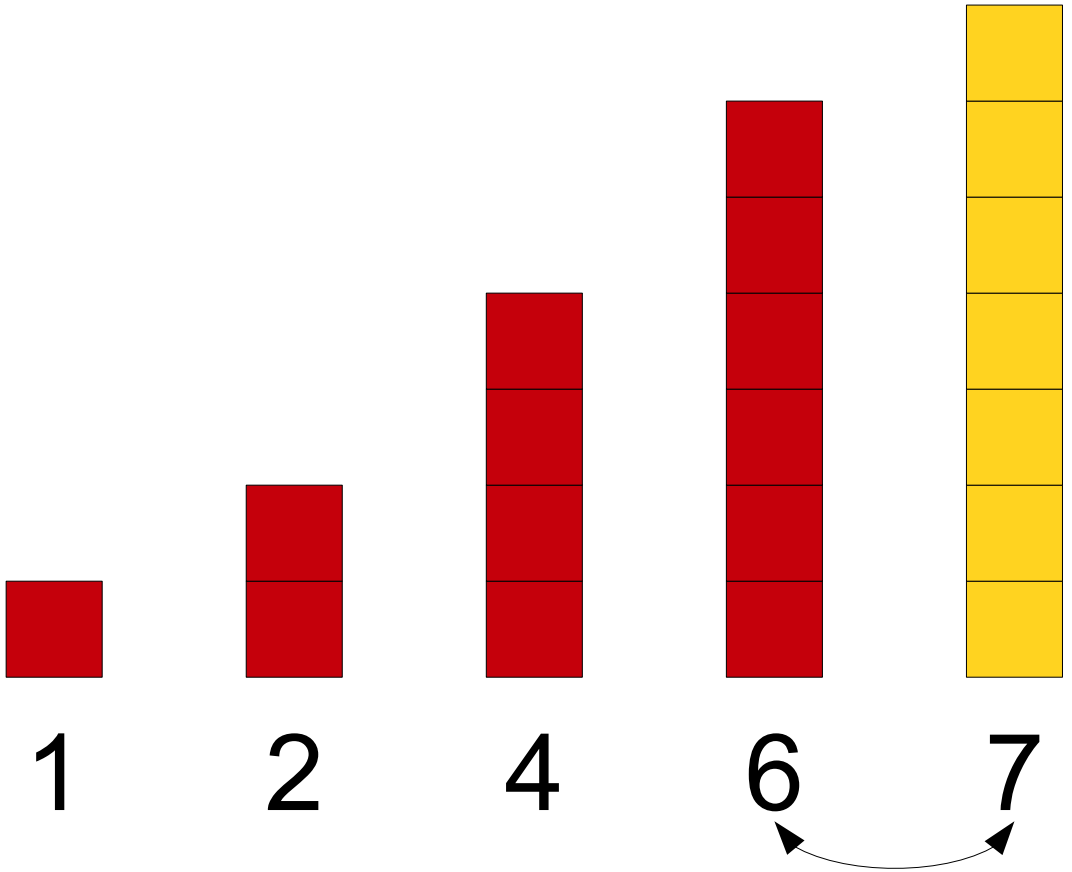
How Fast is Insertion Sort?



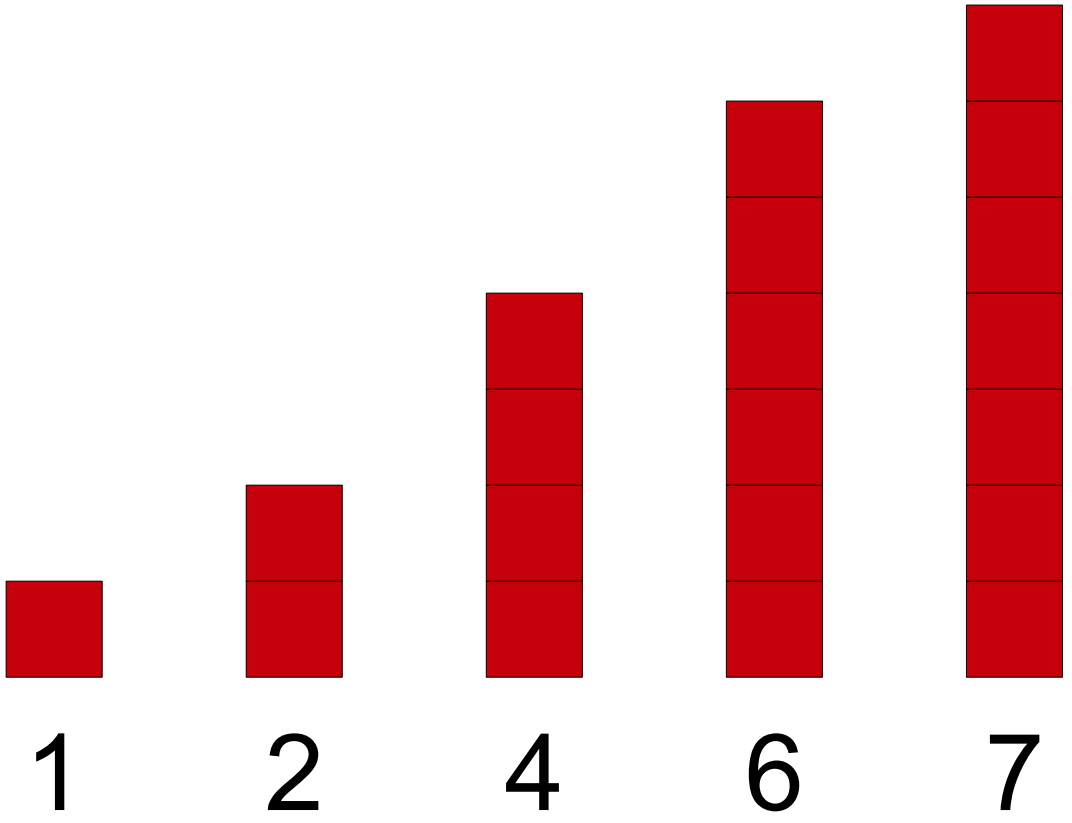
How Fast is Insertion Sort?



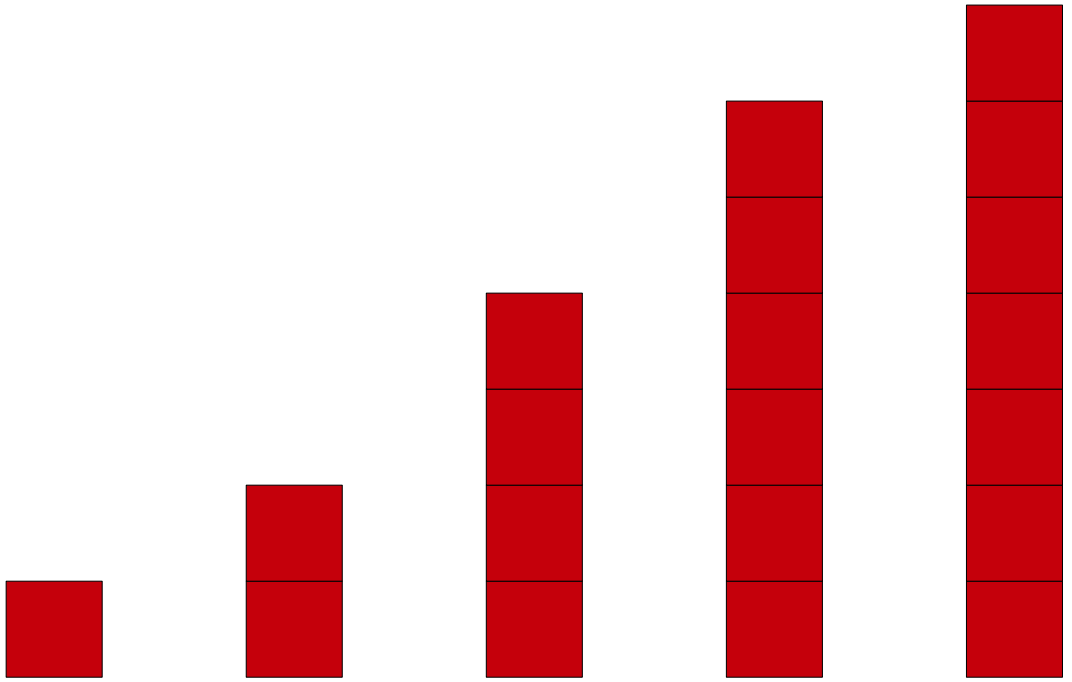
How Fast is Insertion Sort?



How Fast is Insertion Sort?

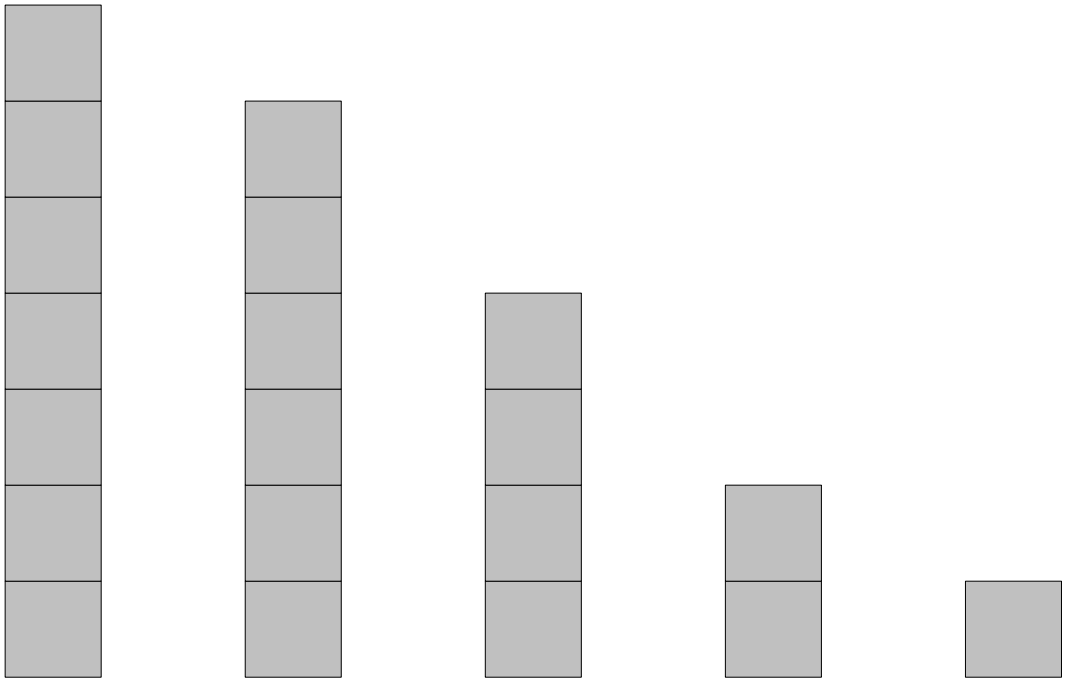


How Fast is Insertion Sort?

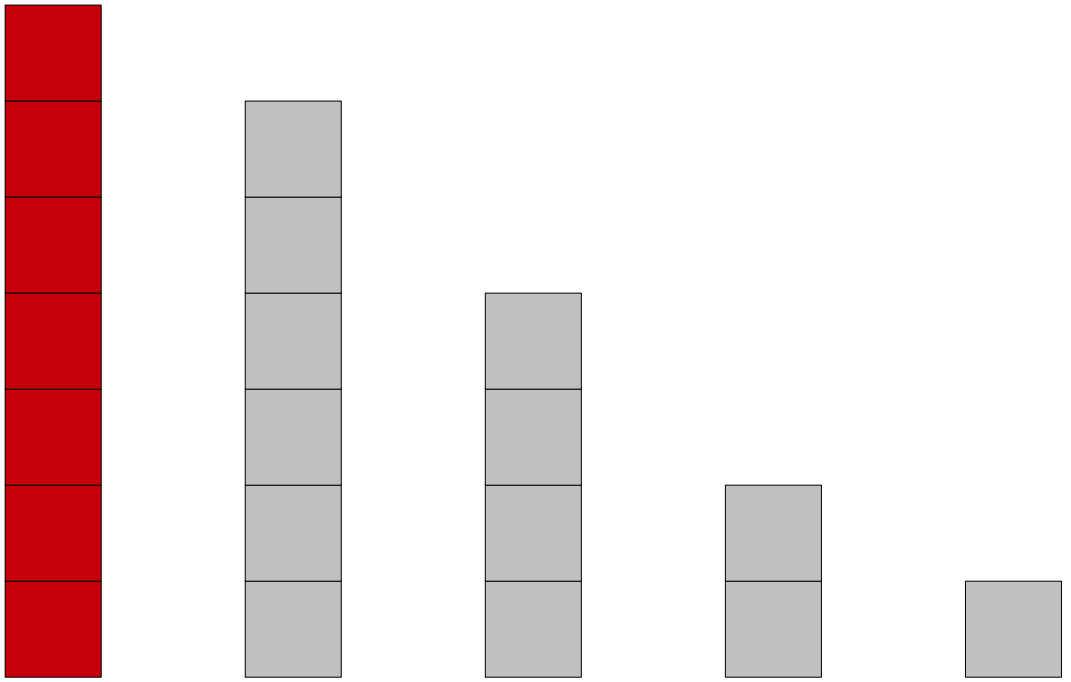


Work done: **O(*n*)**

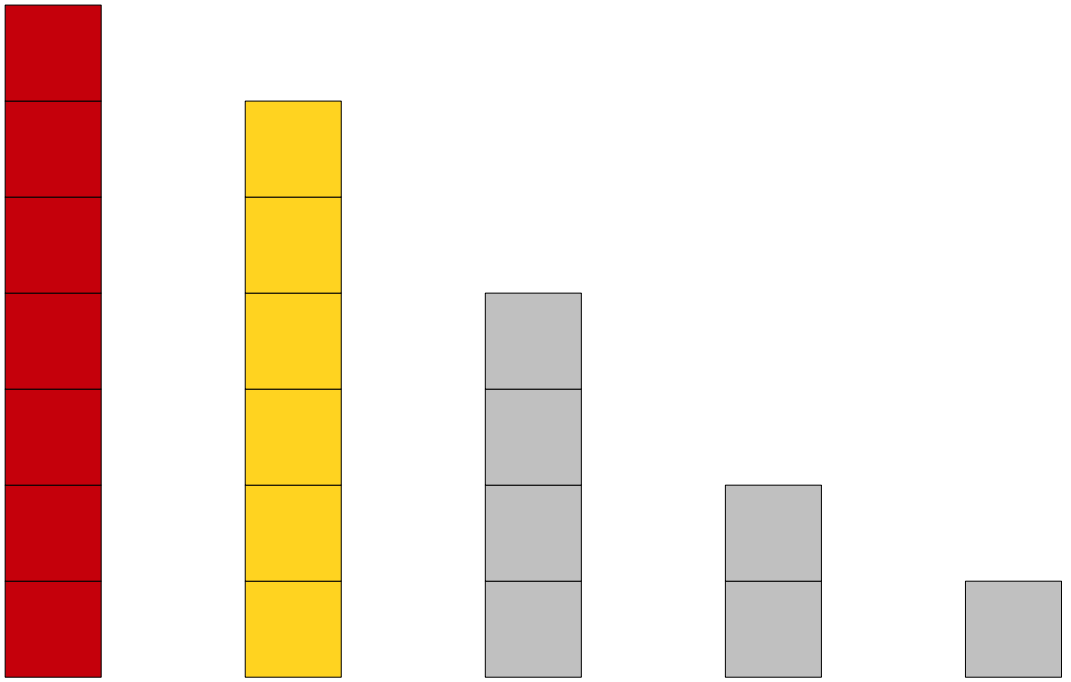
How Fast is Insertion Sort?



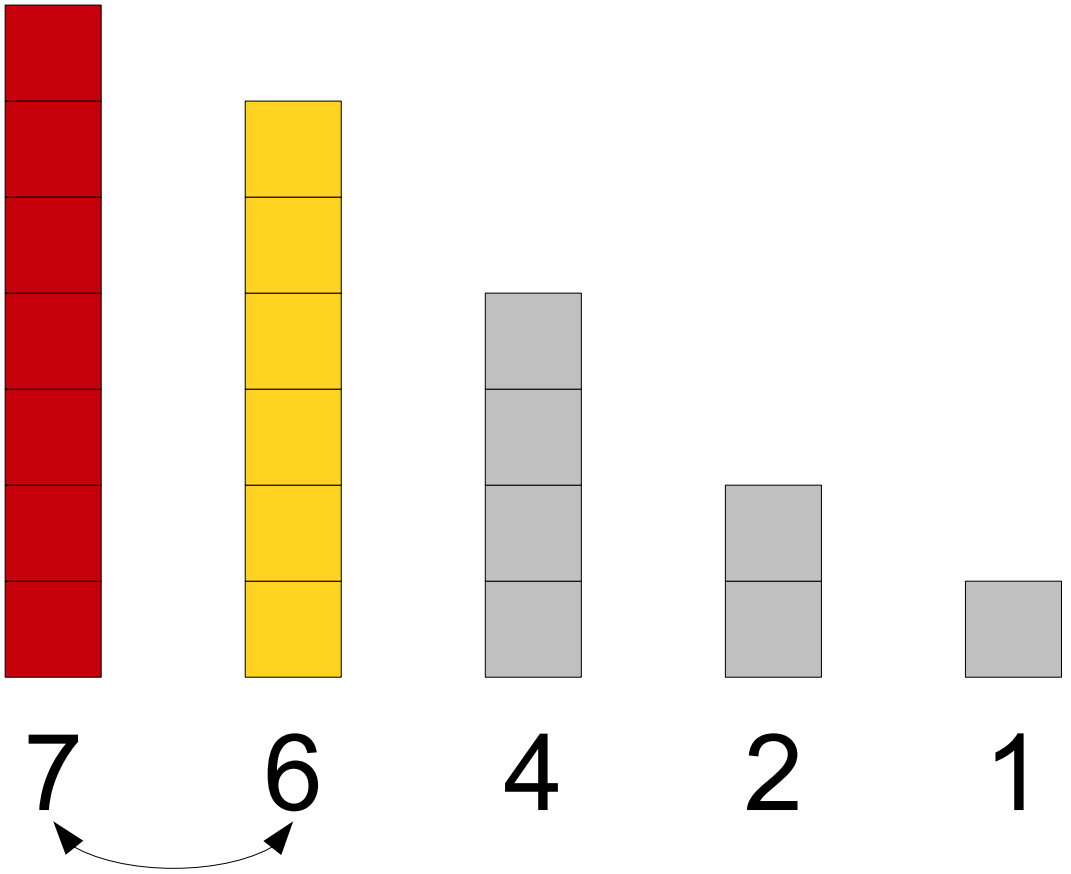
How Fast is Insertion Sort?



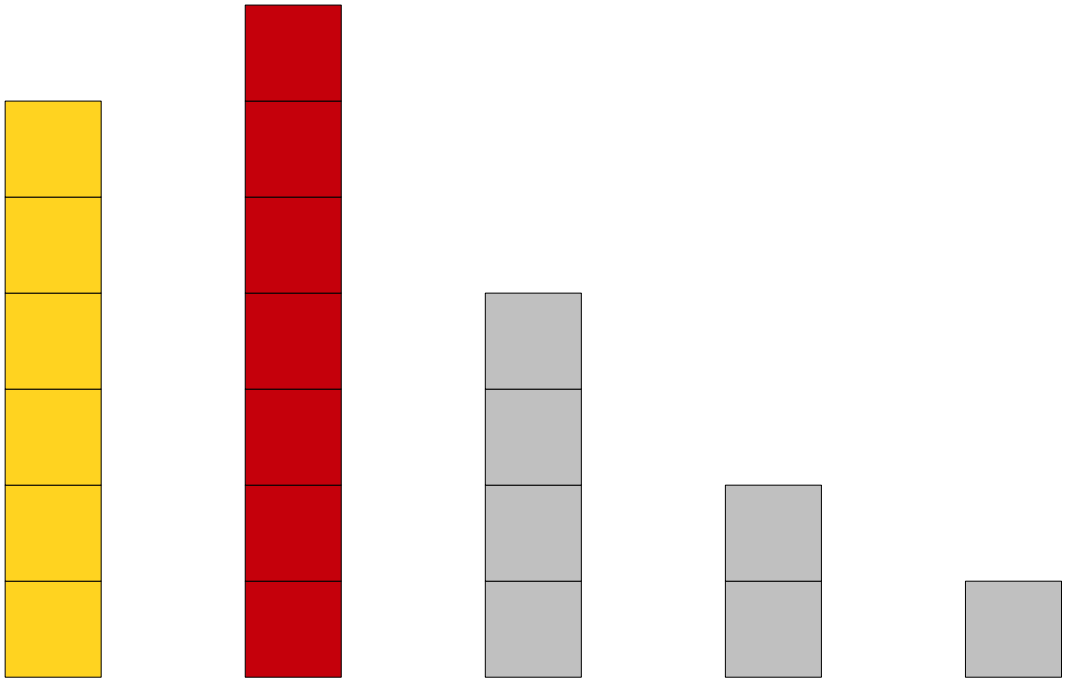
How Fast is Insertion Sort?



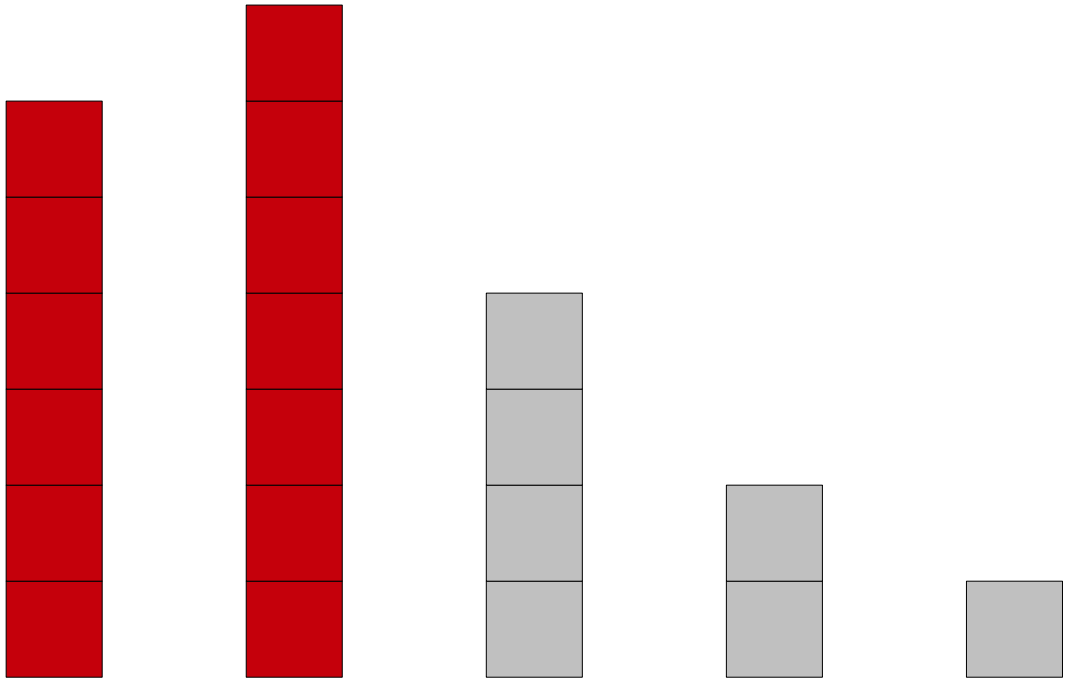
How Fast is Insertion Sort?



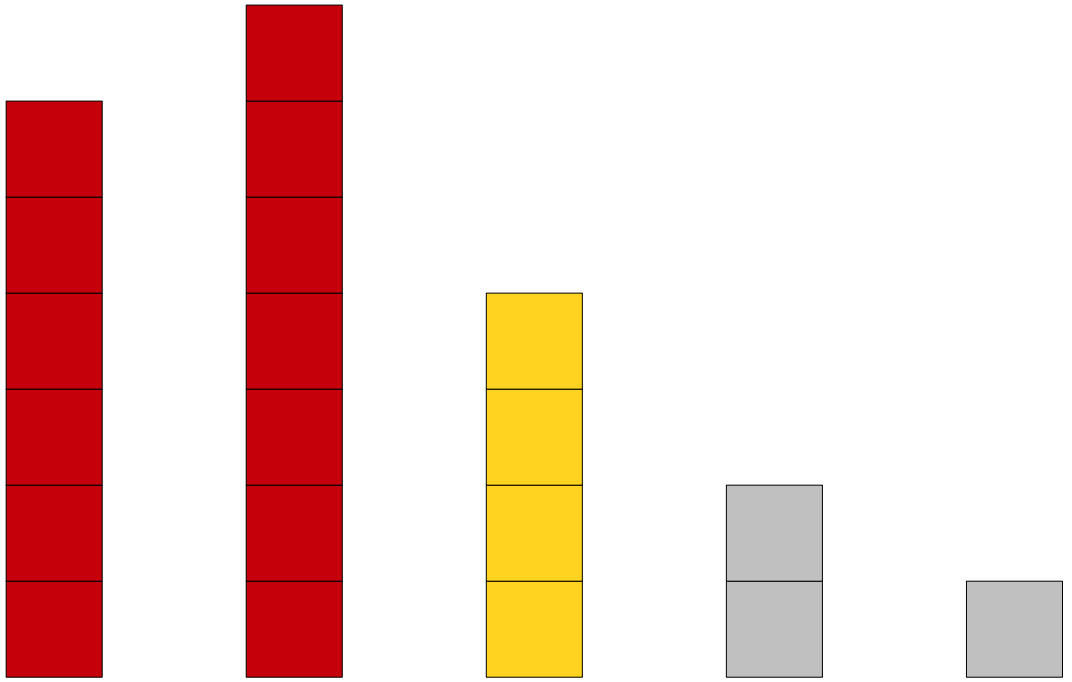
How Fast is Insertion Sort?



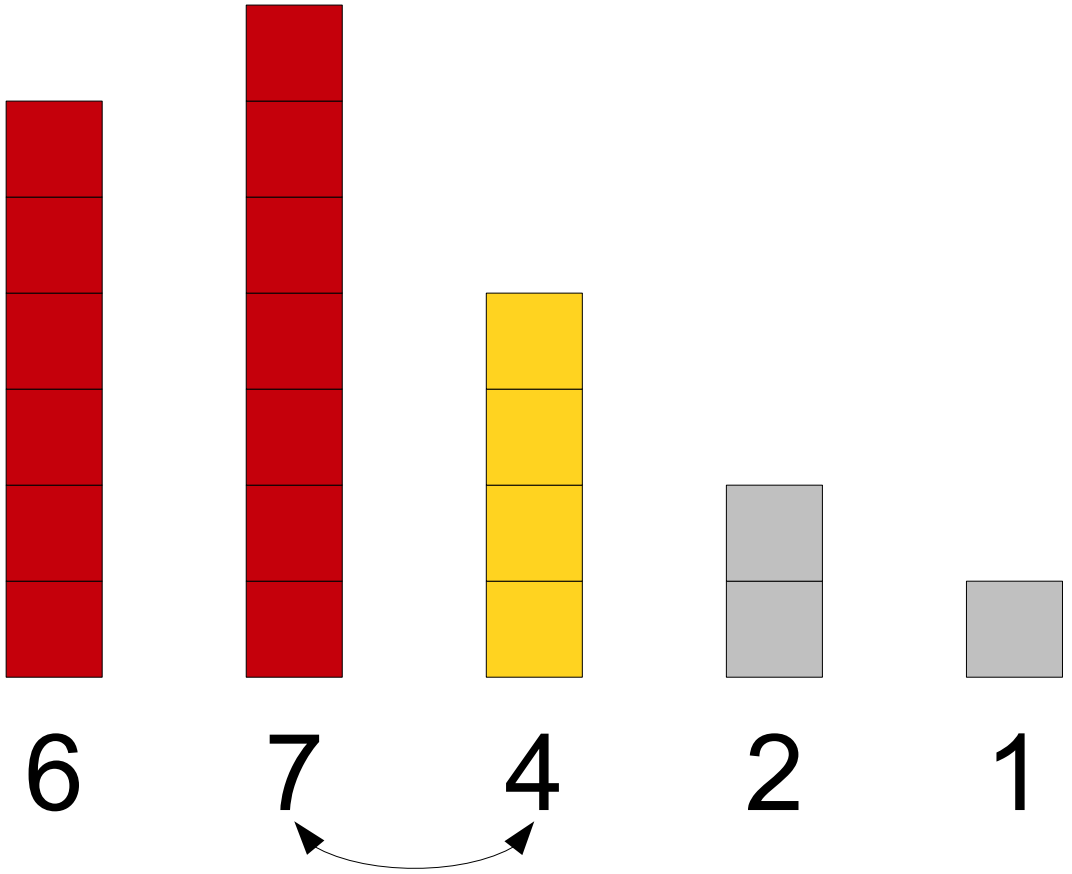
How Fast is Insertion Sort?



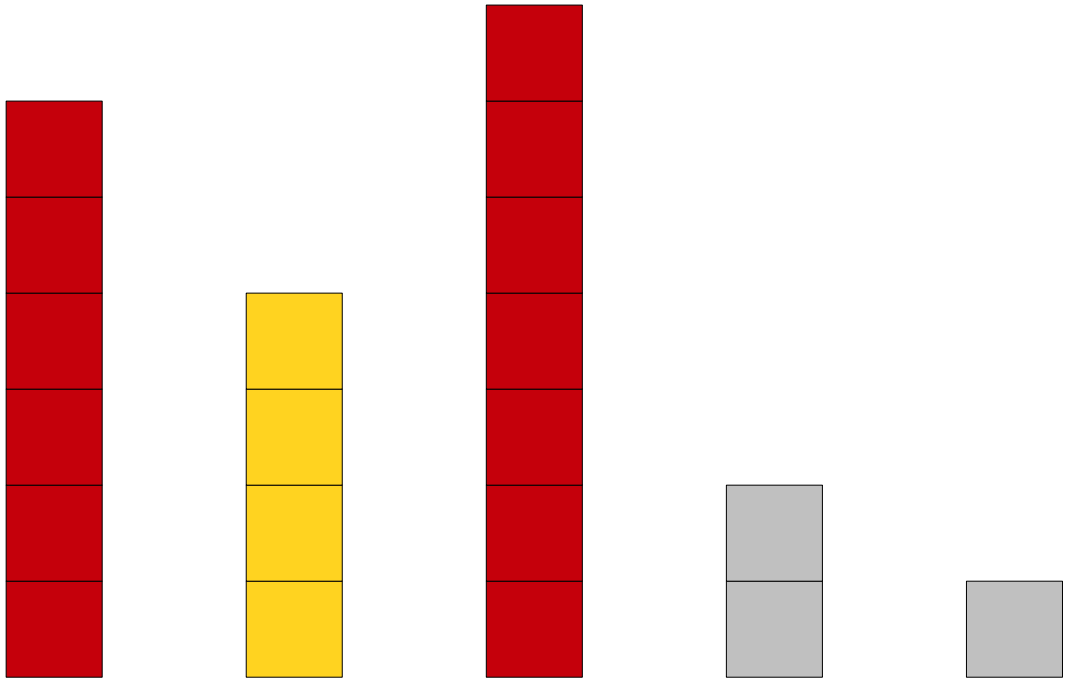
How Fast is Insertion Sort?



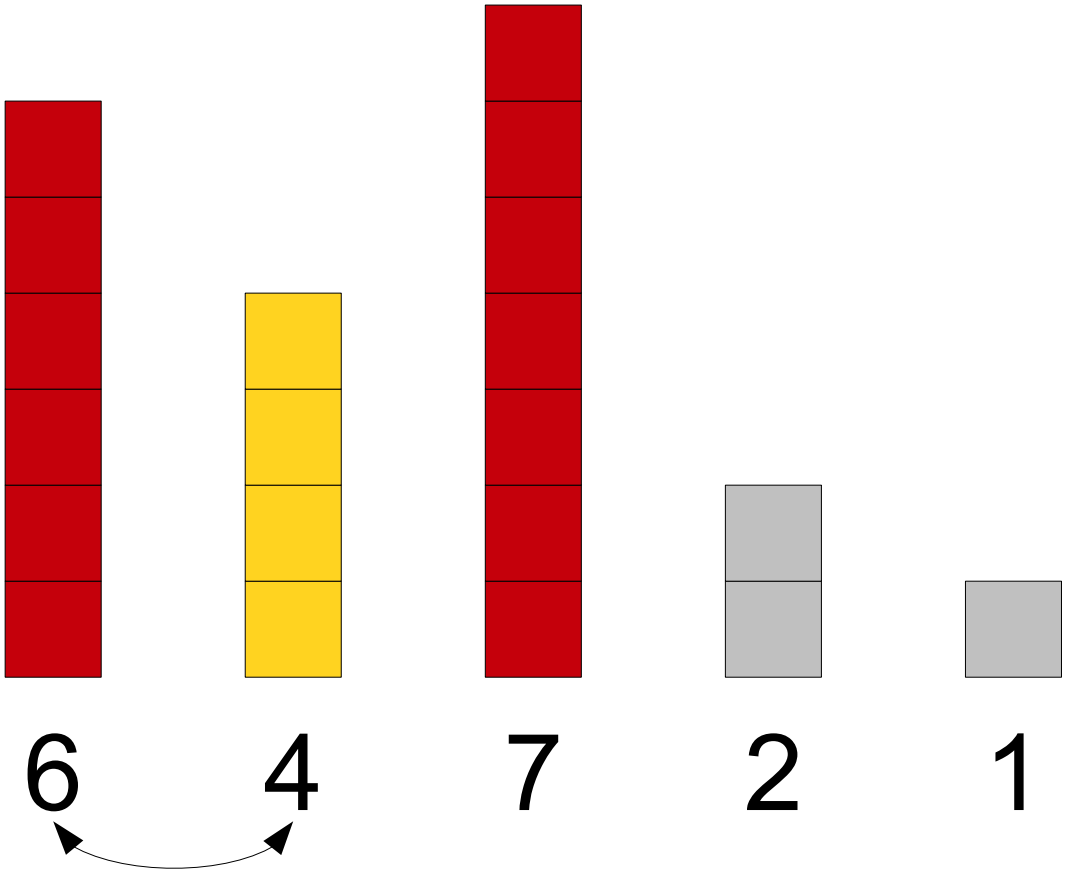
How Fast is Insertion Sort?



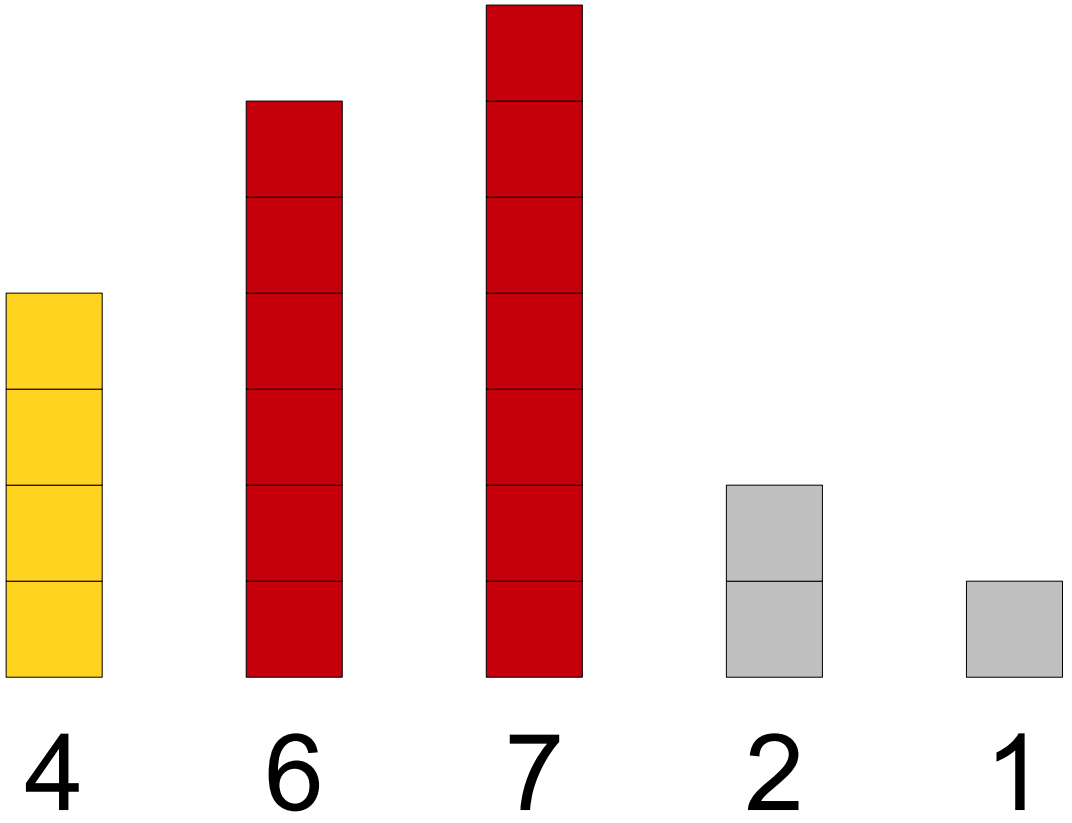
How Fast is Insertion Sort?



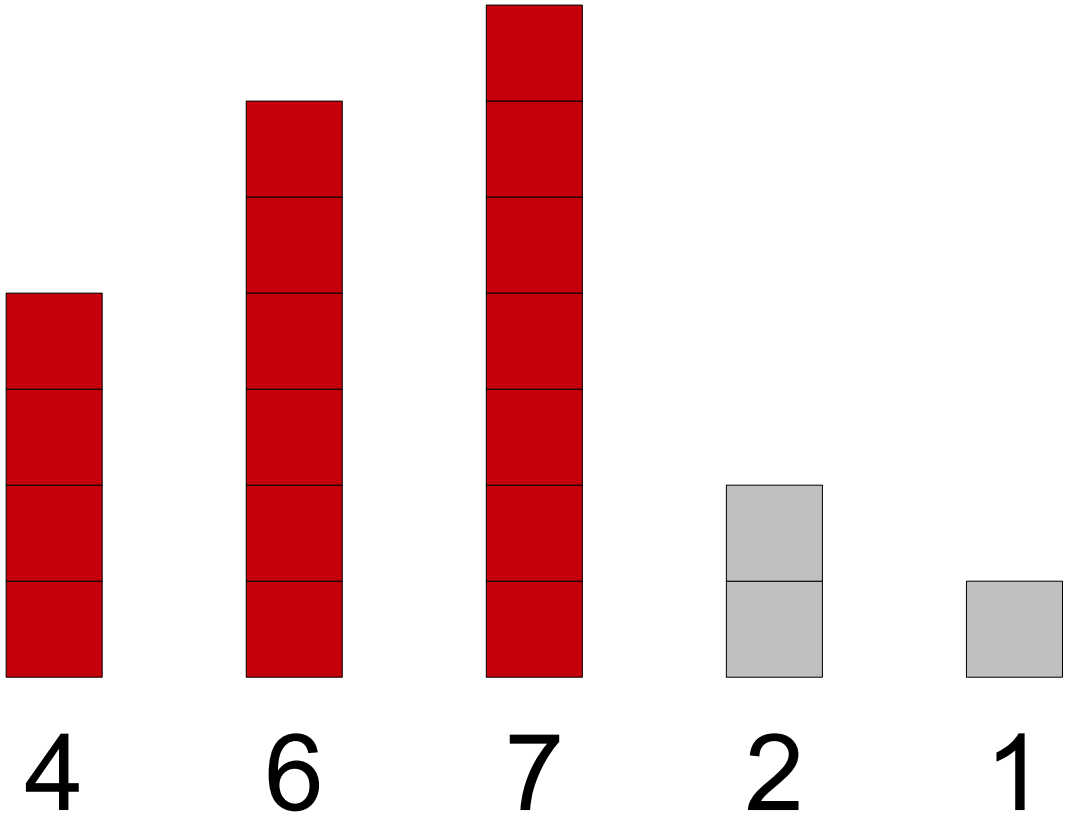
How Fast is Insertion Sort?



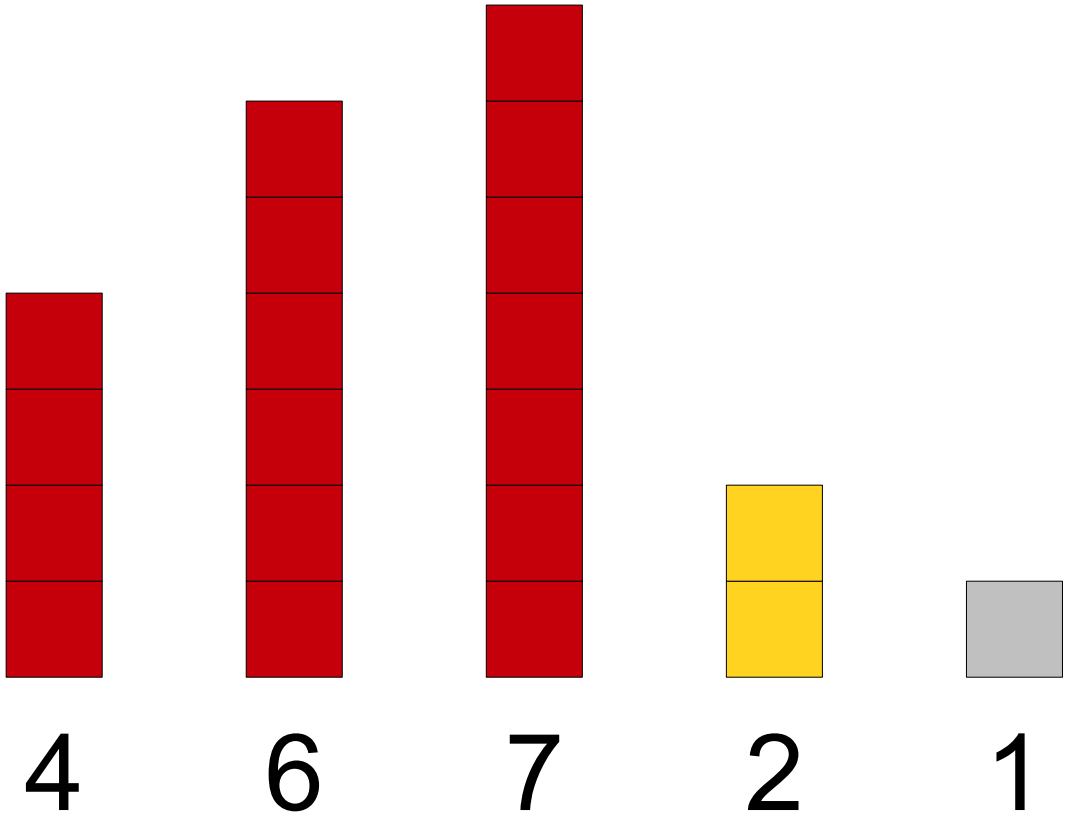
How Fast is Insertion Sort?



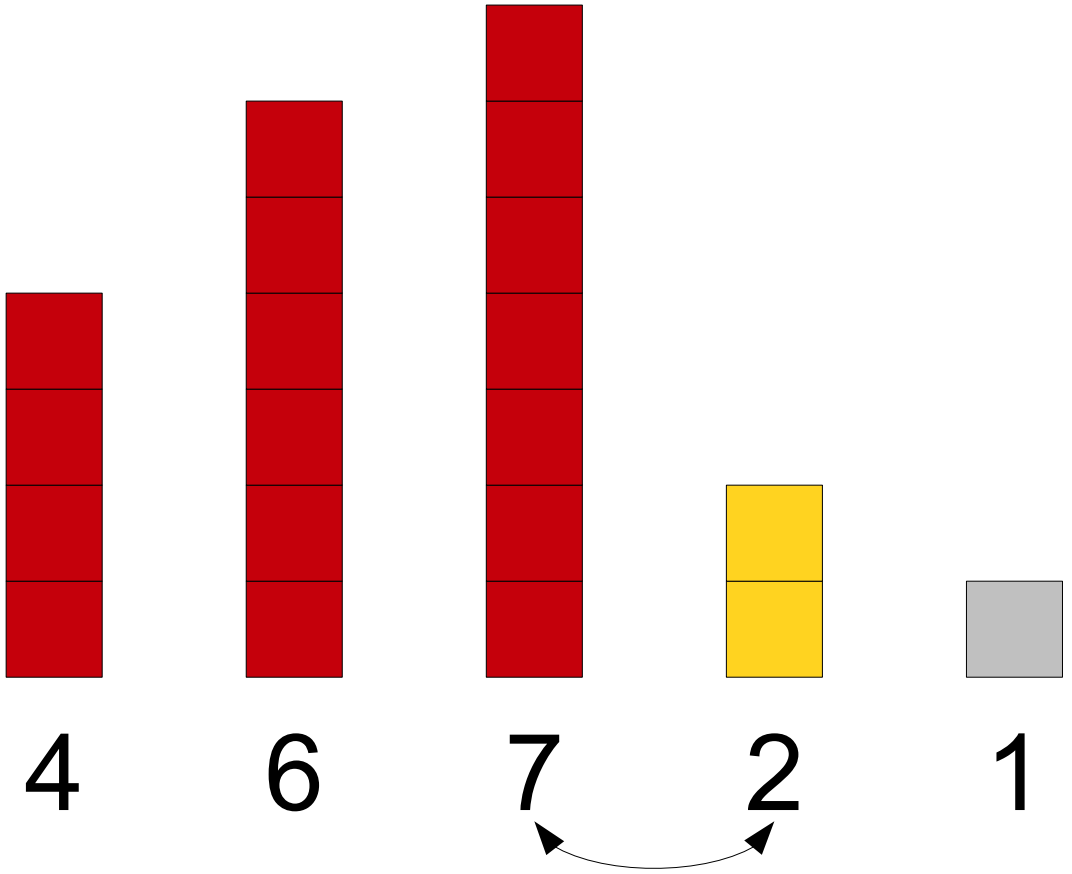
How Fast is Insertion Sort?



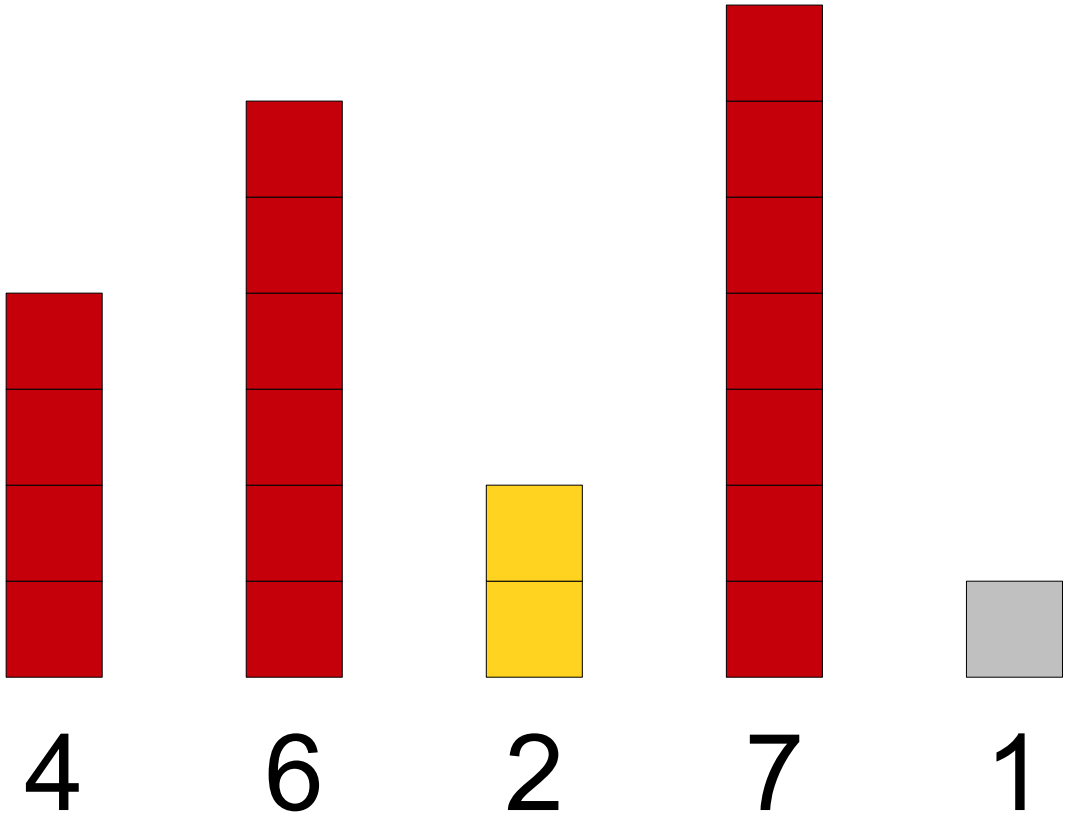
How Fast is Insertion Sort?



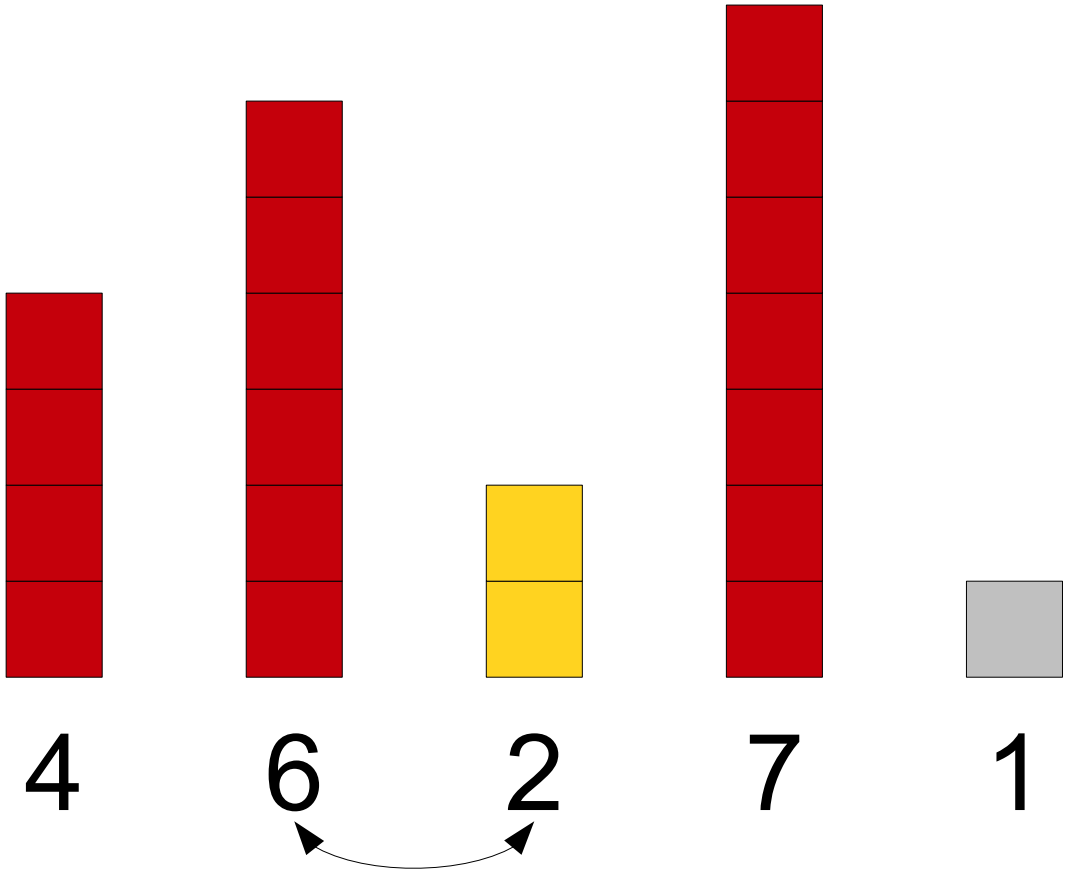
How Fast is Insertion Sort?



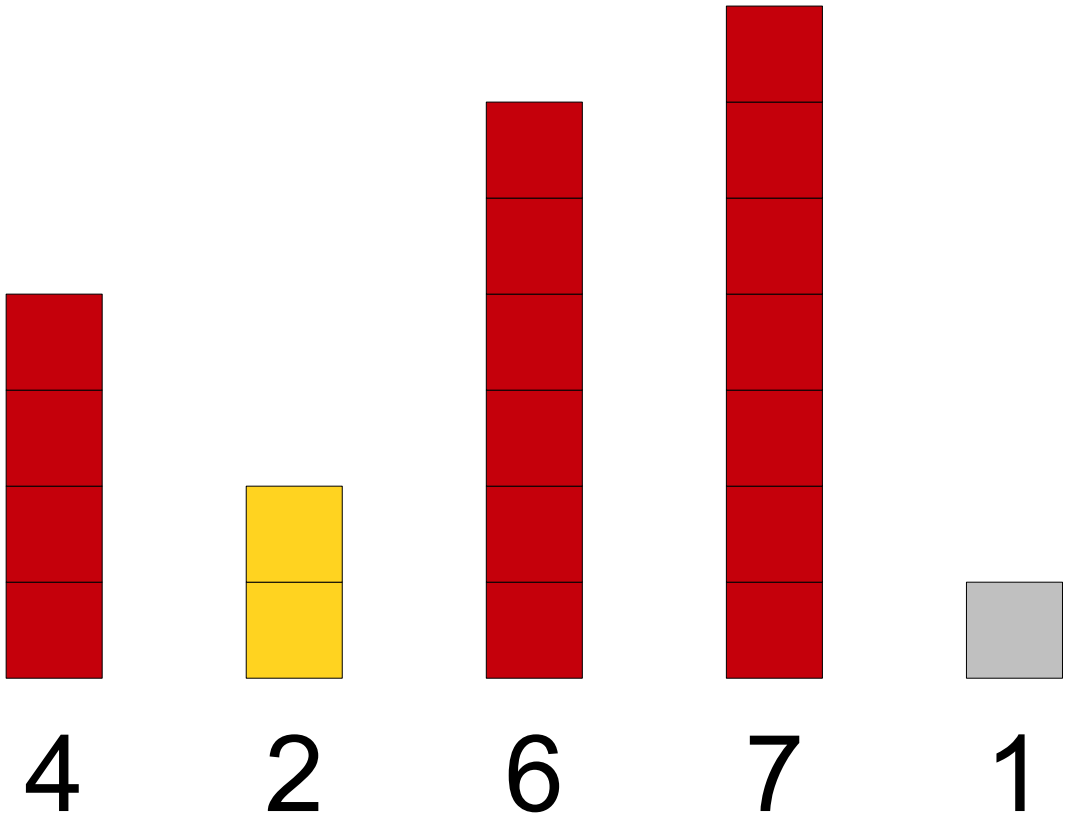
How Fast is Insertion Sort?



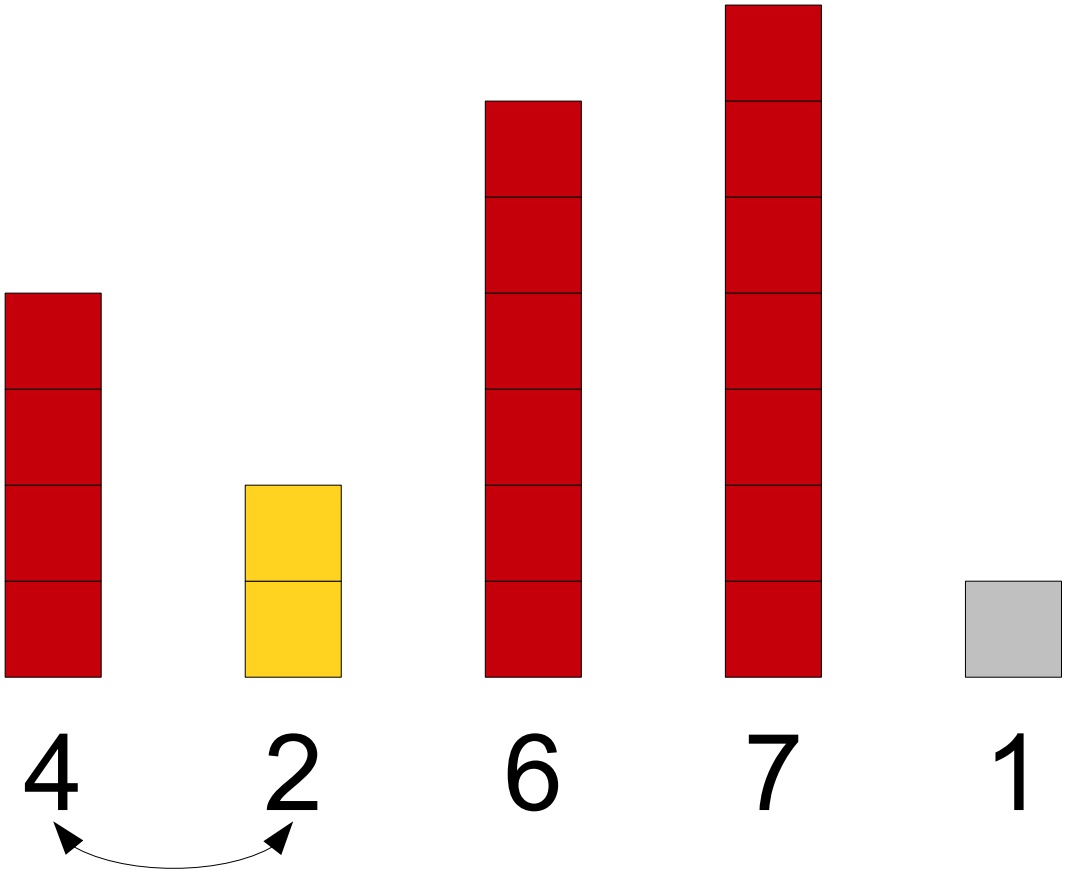
How Fast is Insertion Sort?



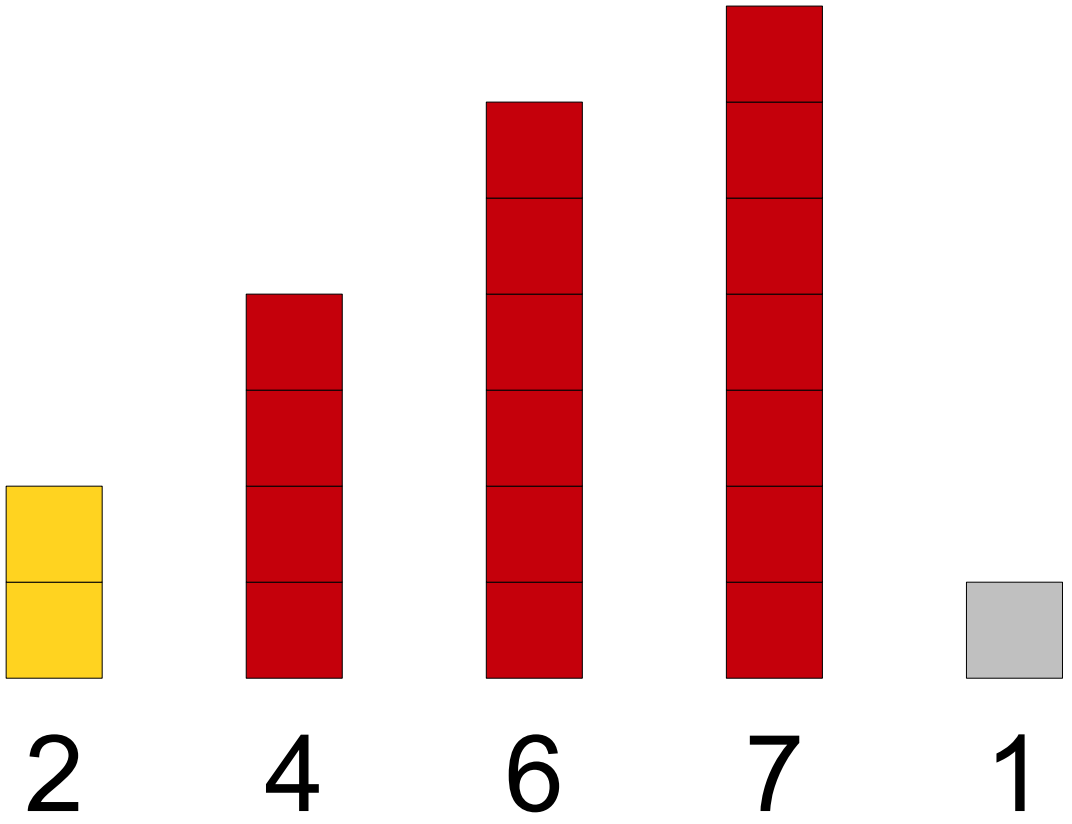
How Fast is Insertion Sort?



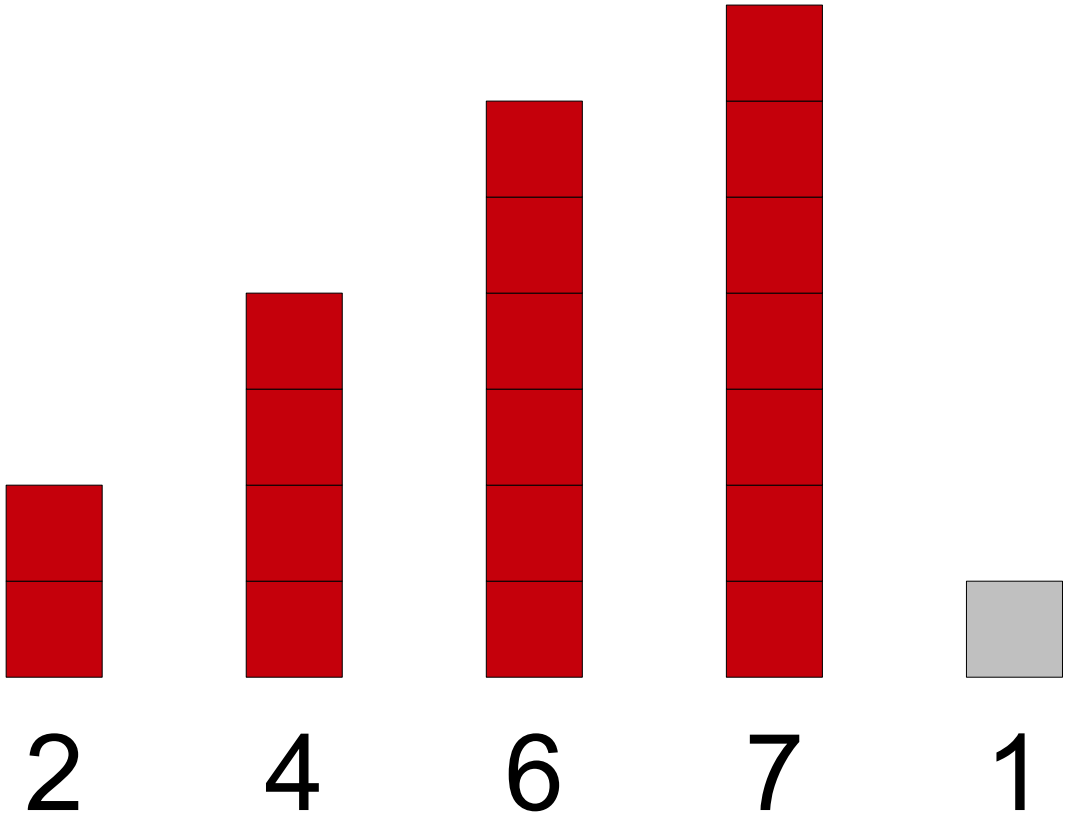
How Fast is Insertion Sort?



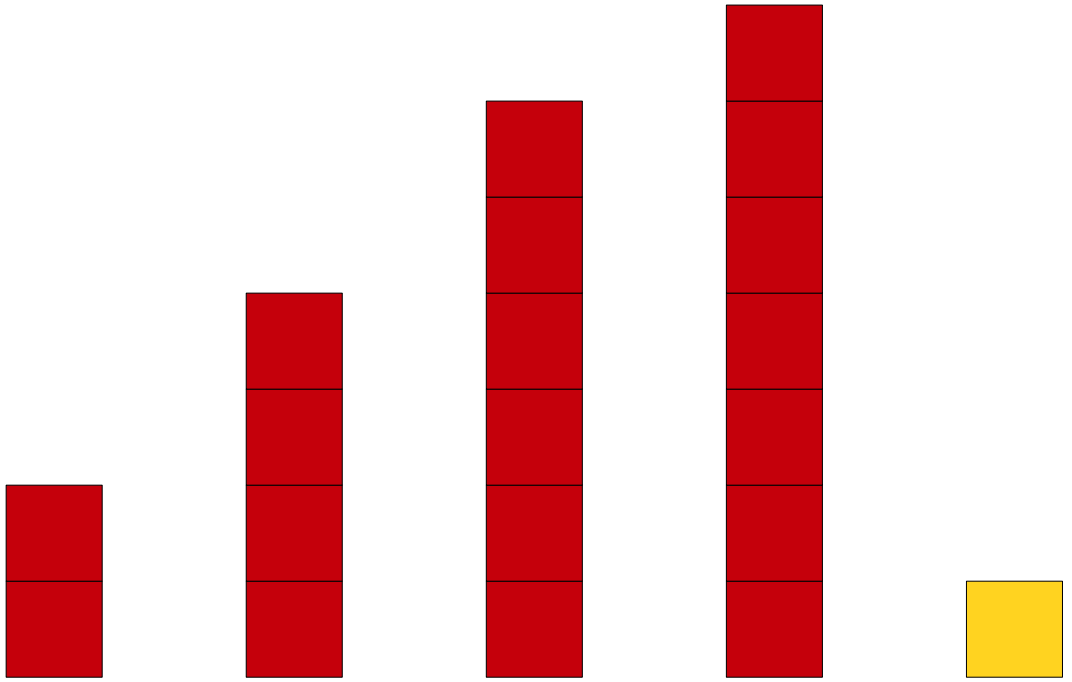
How Fast is Insertion Sort?



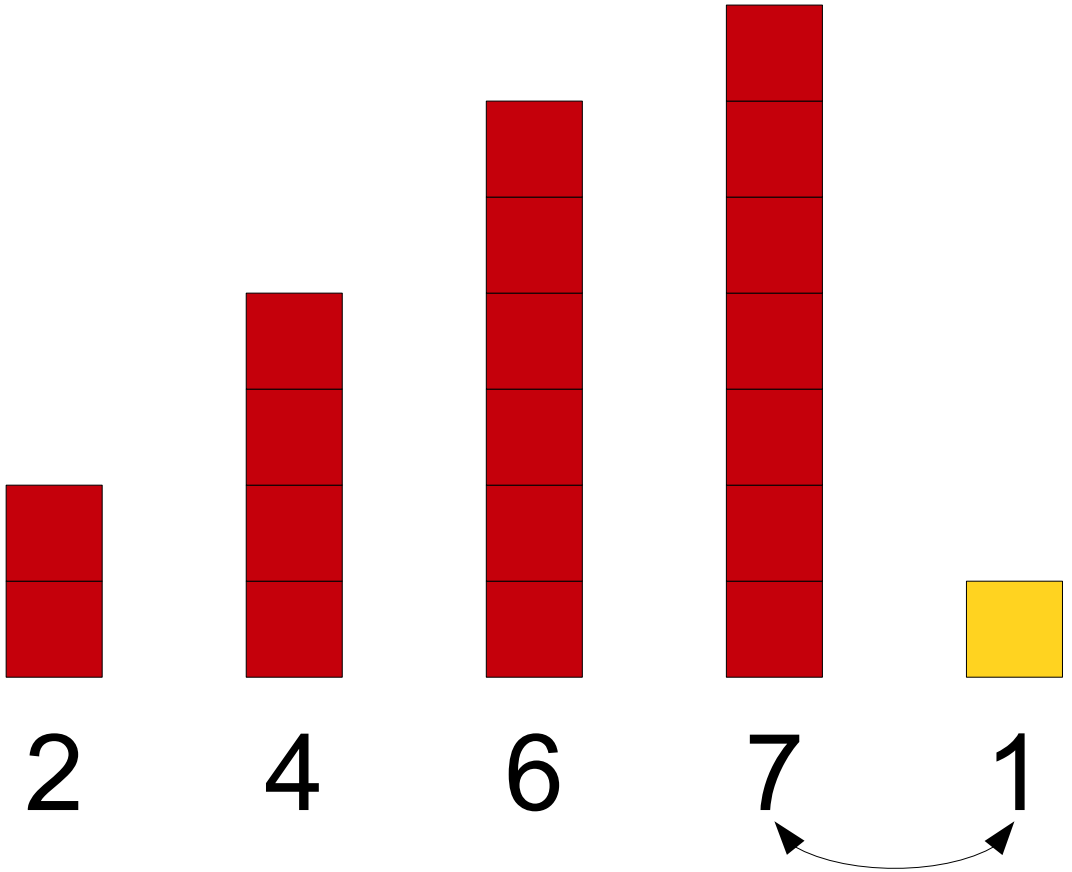
How Fast is Insertion Sort?



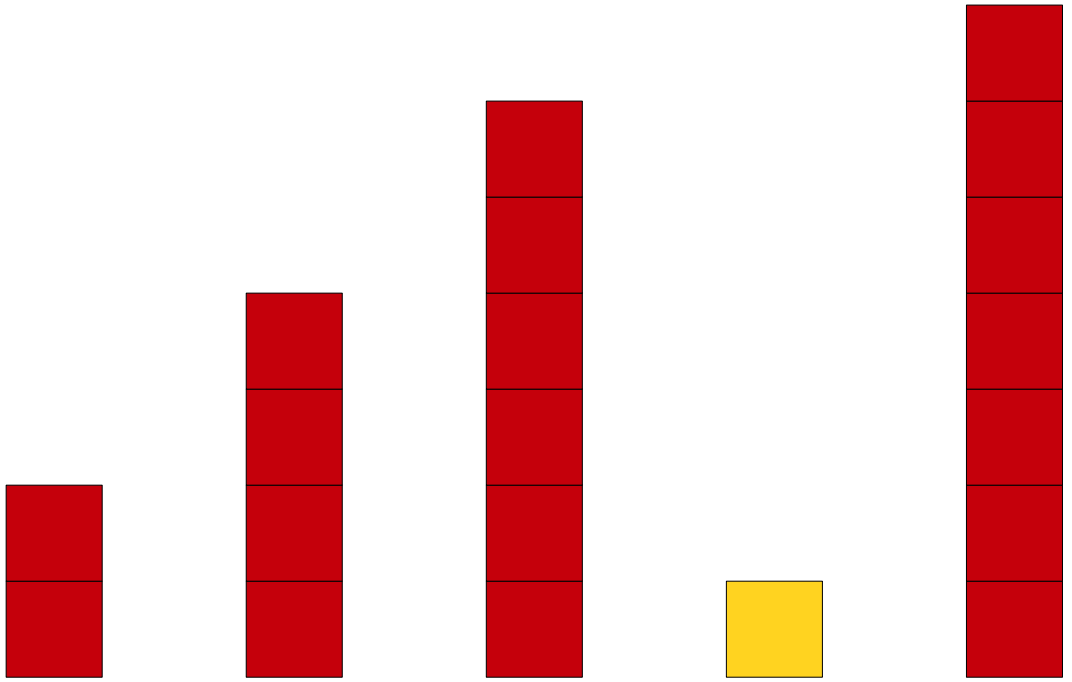
How Fast is Insertion Sort?



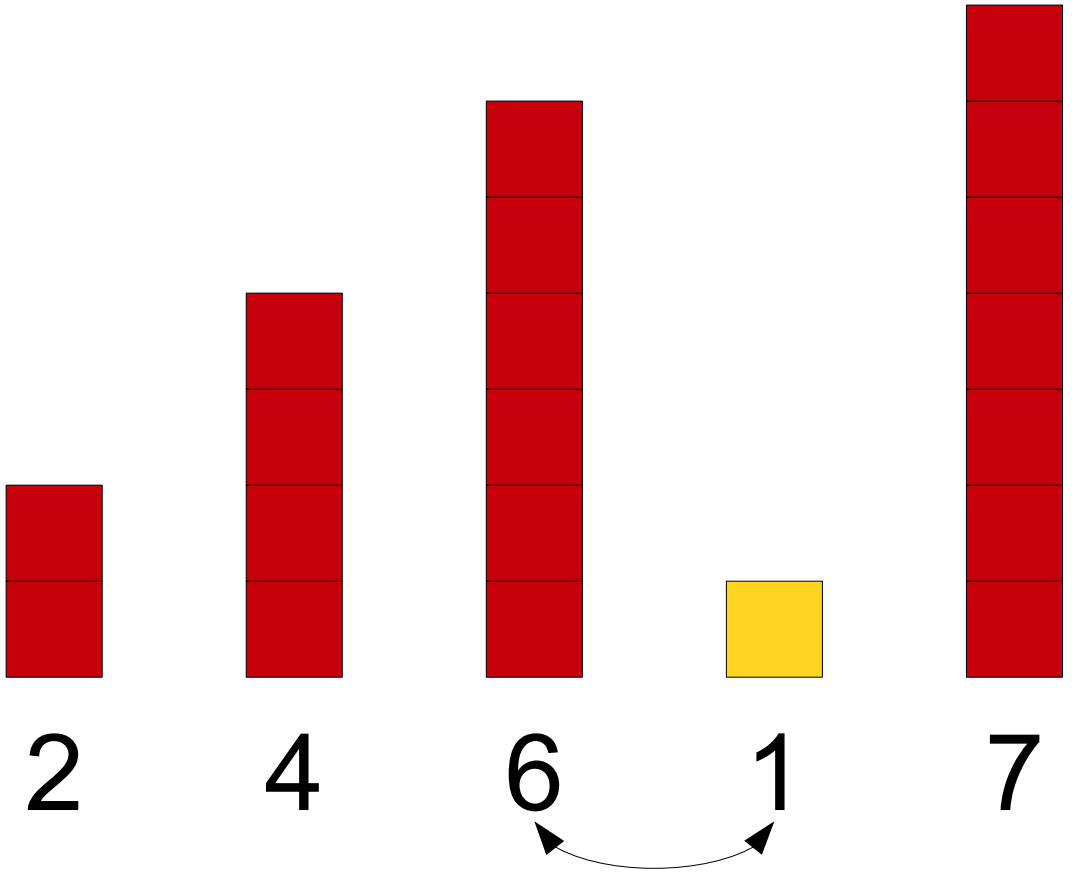
How Fast is Insertion Sort?



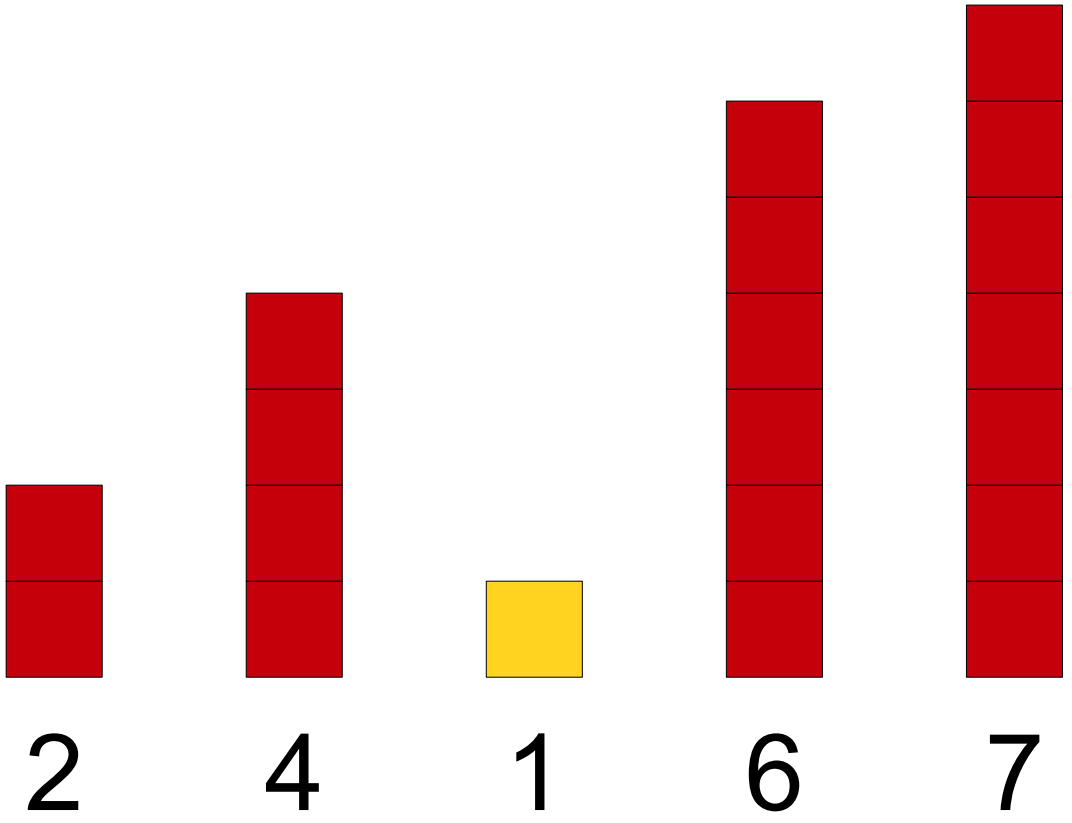
How Fast is Insertion Sort?



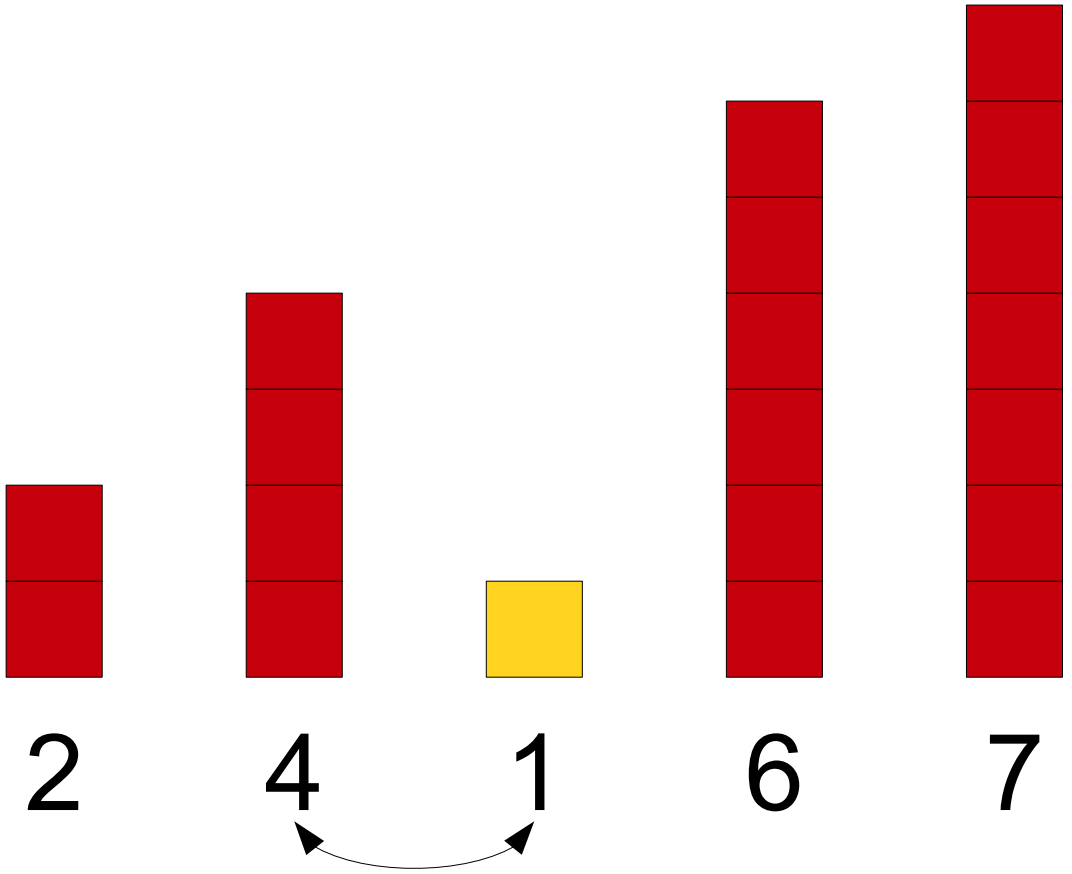
How Fast is Insertion Sort?



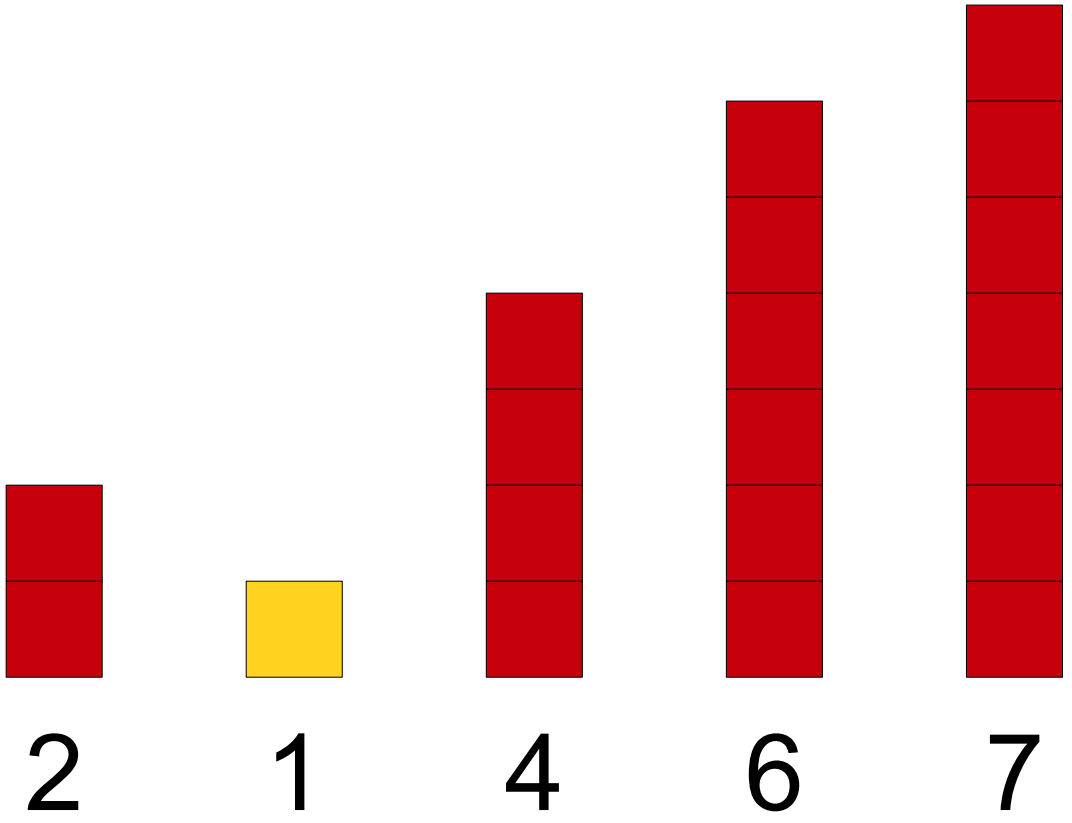
How Fast is Insertion Sort?



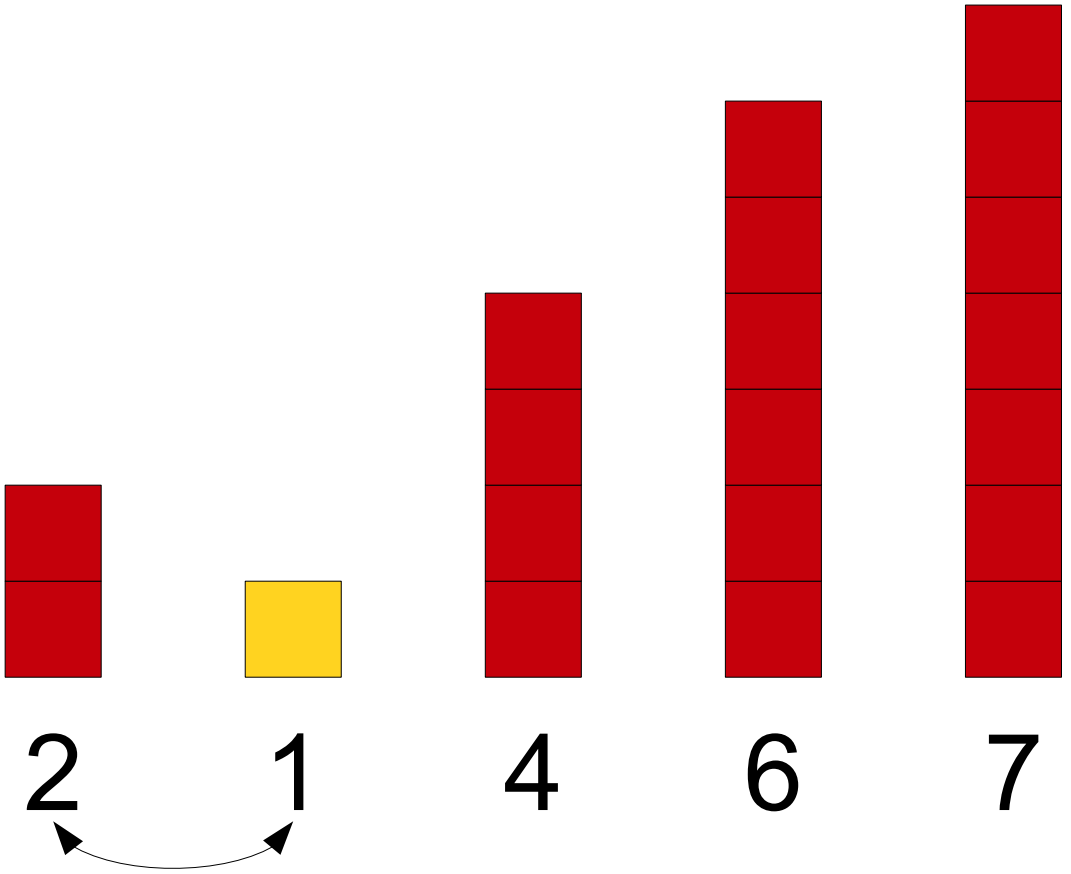
How Fast is Insertion Sort?



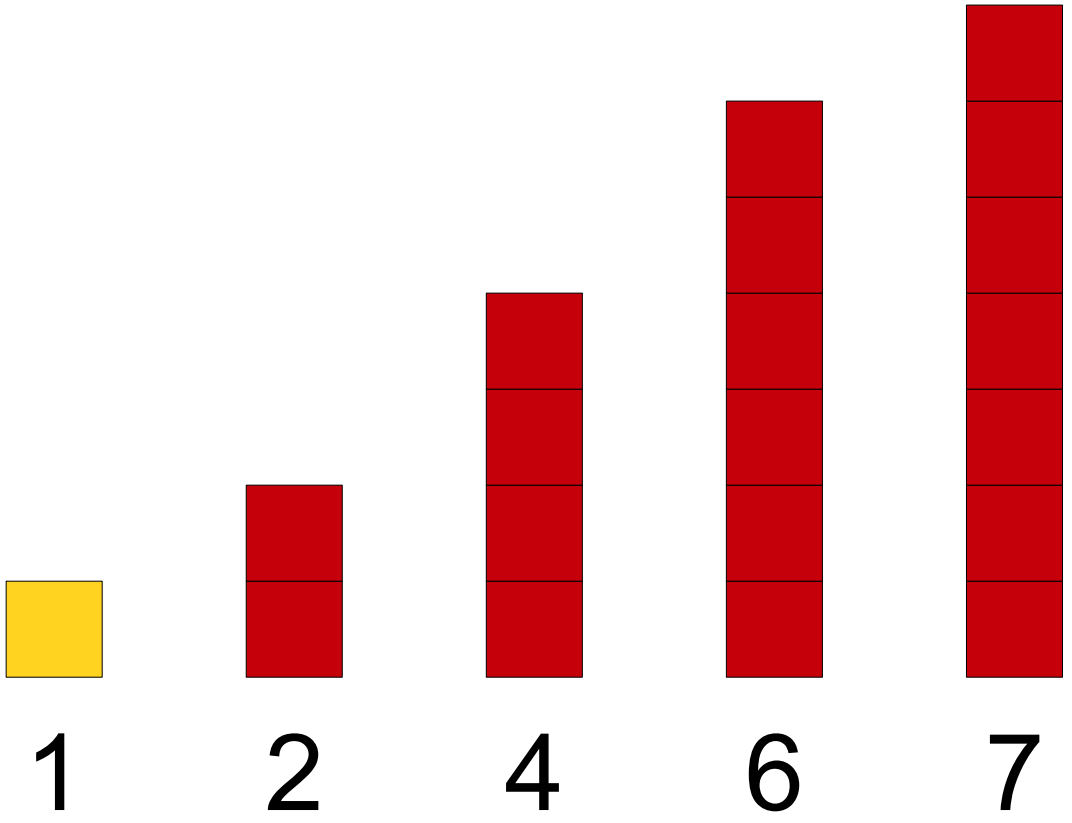
How Fast is Insertion Sort?



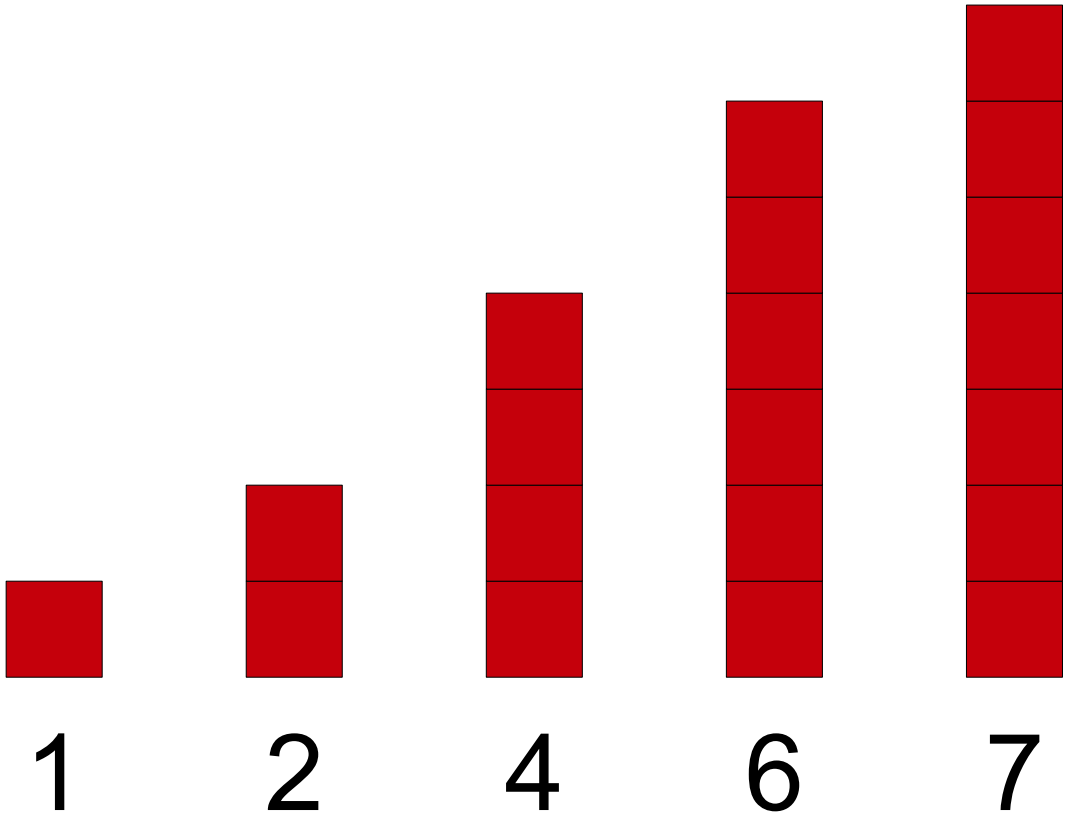
How Fast is Insertion Sort?



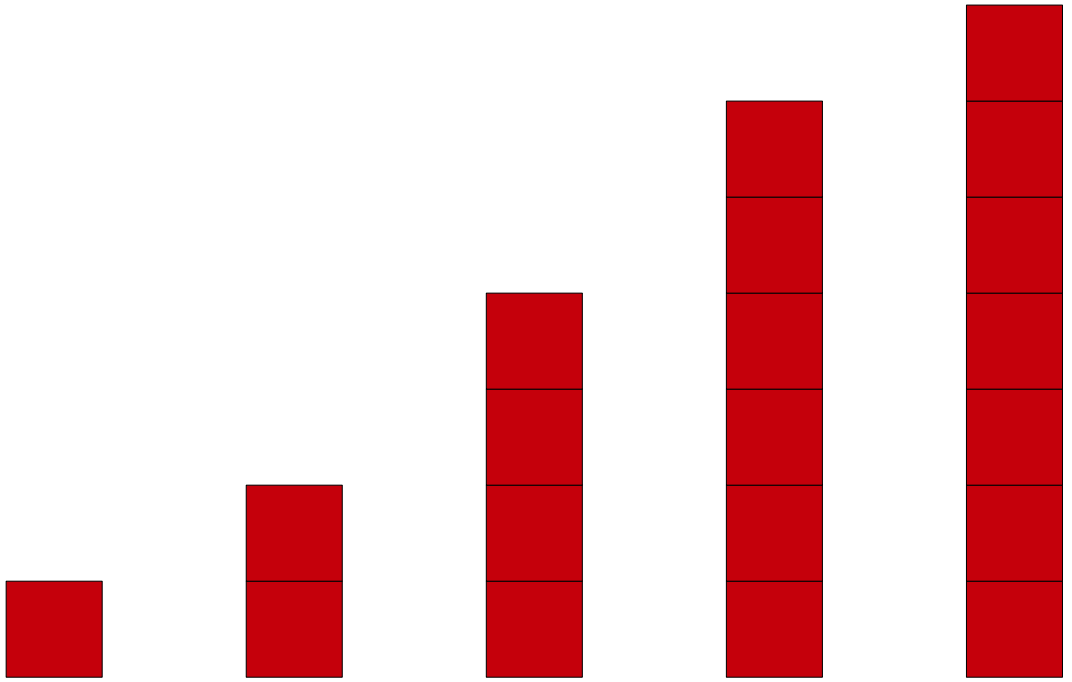
How Fast is Insertion Sort?



How Fast is Insertion Sort?



How Fast is Insertion Sort?



Work done: **$O(n^2)$**

Notes on Insertion Sort

- Insertion sort has runtime O(n) in the best case.
- Insertion sort has runtime O(n^2) in the worst case.
- On average, insertion sort is roughly twice as fast as selection sort.
  - In a random array, every element is about halfway away from where it belongs.
  - So for the kth element, about (k – 1) / 2 other elements must be looked at.

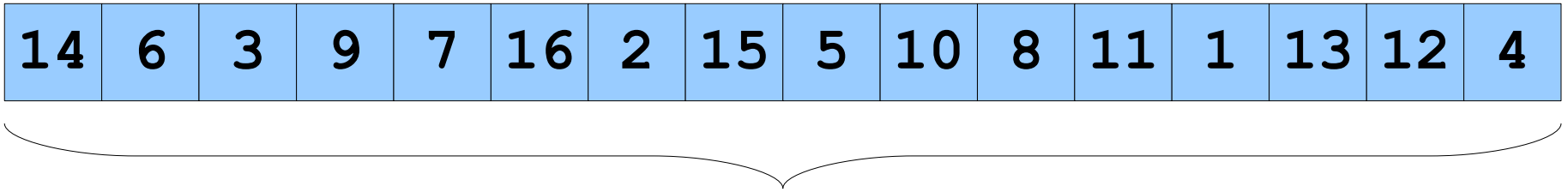
Selection Sort vs Insertion Sort

Size	Selection Sort	Insertion Sort
10000	0.304	0.160
20000	1.218	0.630
30000	2.790	1.427
40000	4.646	2.520
50000	7.395	4.181
60000	10.584	5.635
70000	14.149	8.143
80000	18.674	10.333
90000	23.165	12.832

Of insertion sort and selection sort:

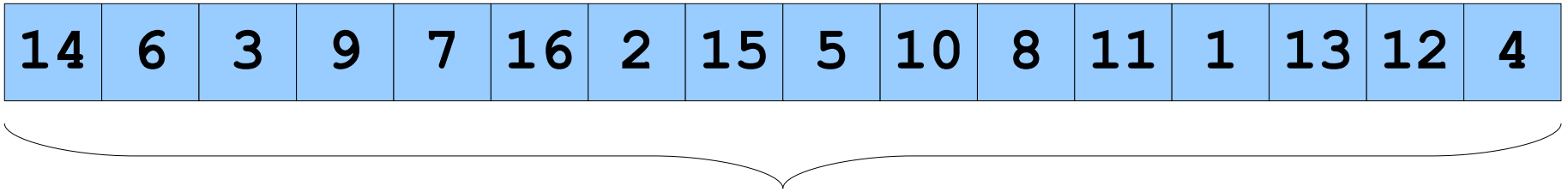
1. Which algorithm does more work at the start?
2. Which algorithm does more work at the end?

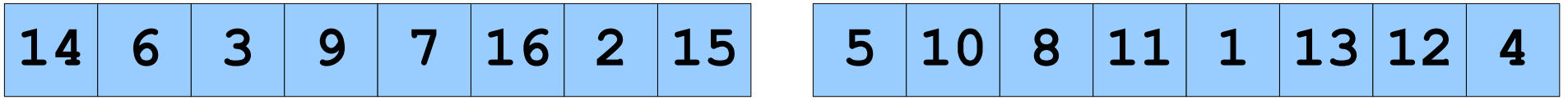
Thinking About O(n^2)



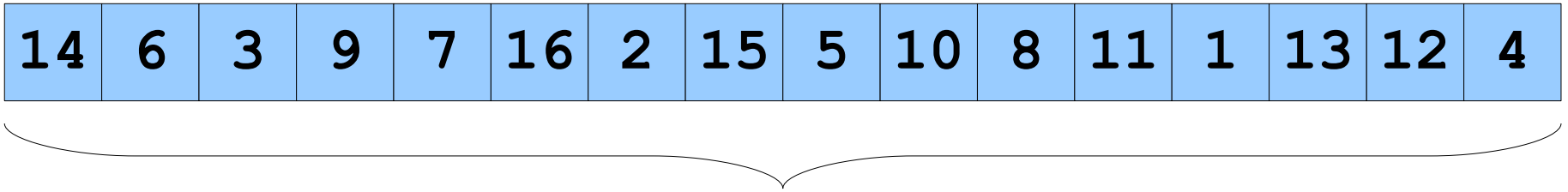
T(n)

Thinking About $O(n^2)$

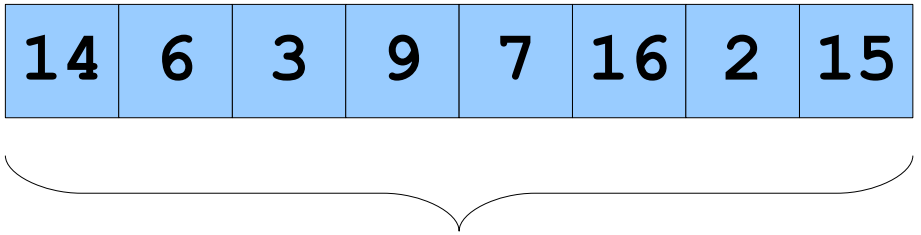




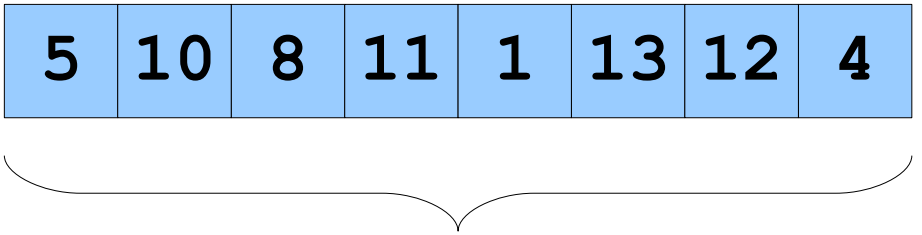
Thinking About O(n^2)



T(n)

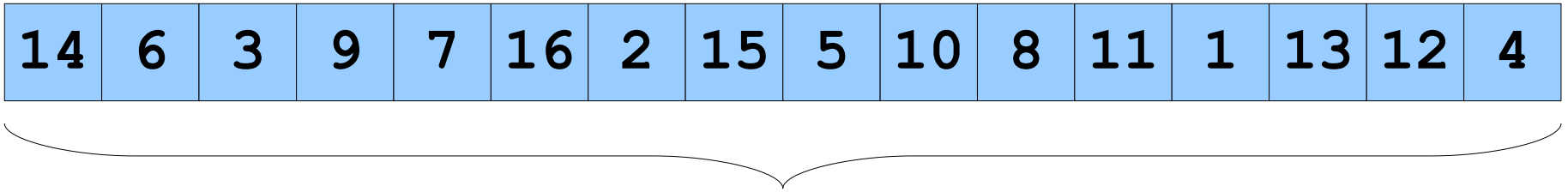


T(½n) ≈ ¼T(n)

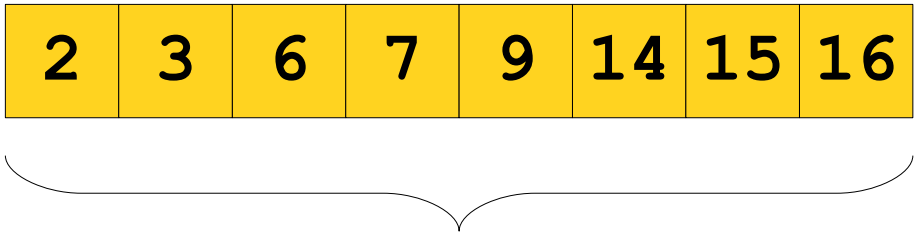


T(½n) ≈ ¼T(n)

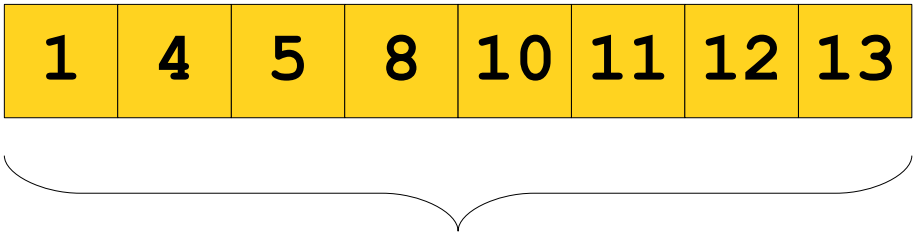
Thinking About O(n^2)



T(n)

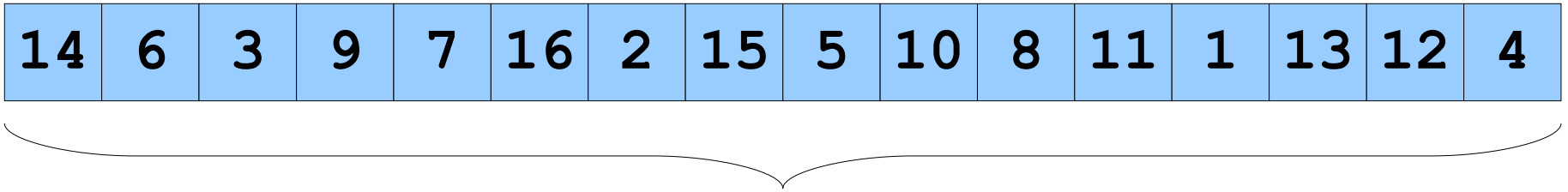


T(½n) ≈ ¼T(n)

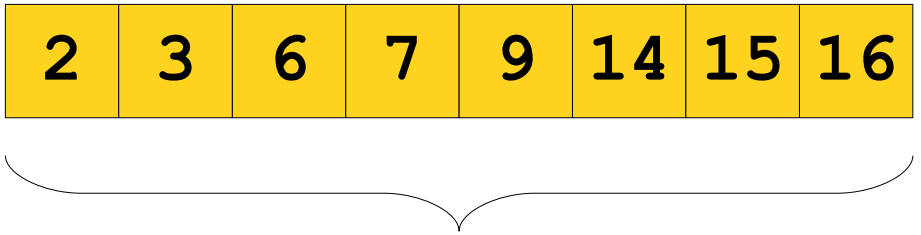


T(½n) ≈ ¼T(n)

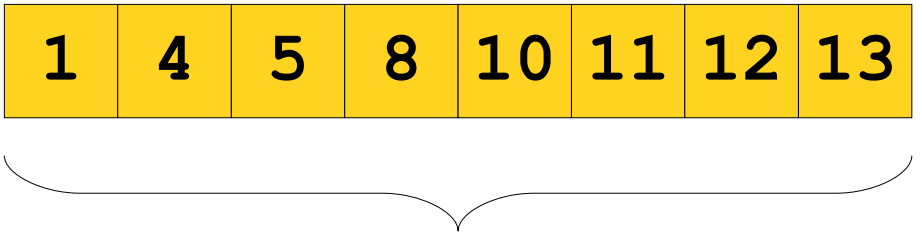
Thinking About O(n^2)



$T(n)$



$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$

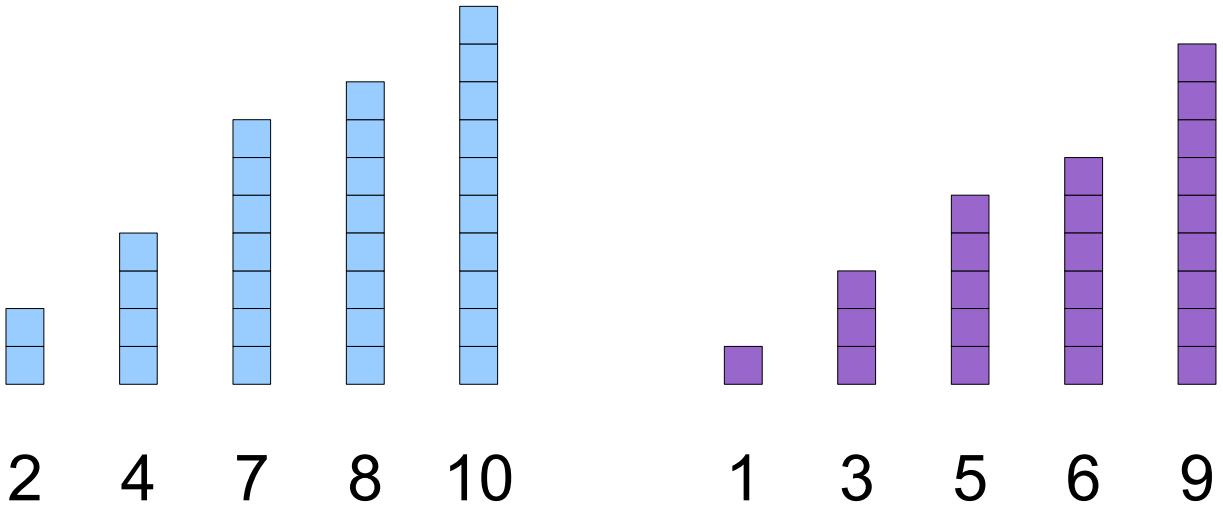


$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$

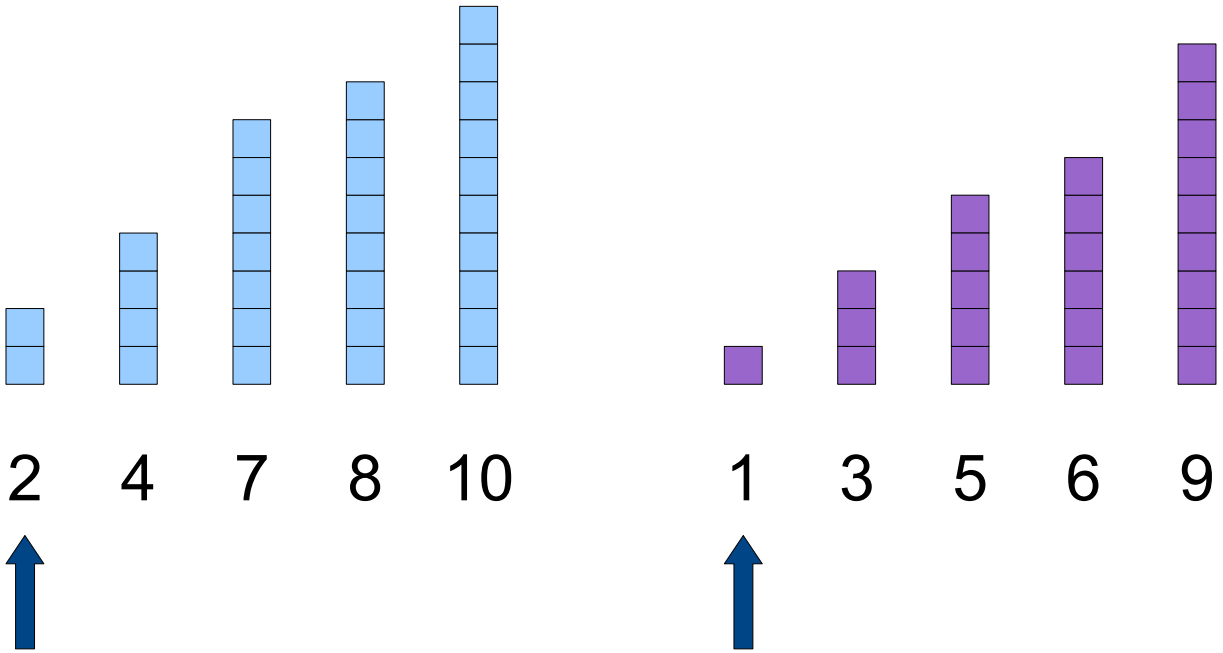
It takes roughly ½T(n) to sort each half separately!

The Key Insight: **Merge**

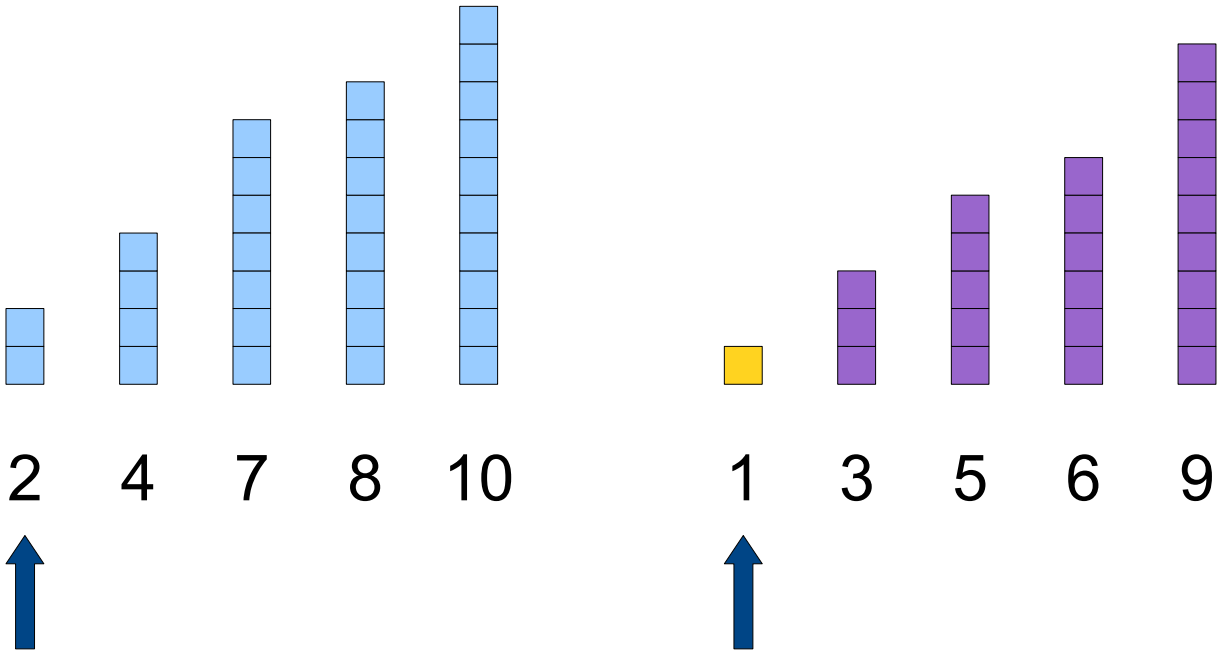
The Key Insight: **Merge**



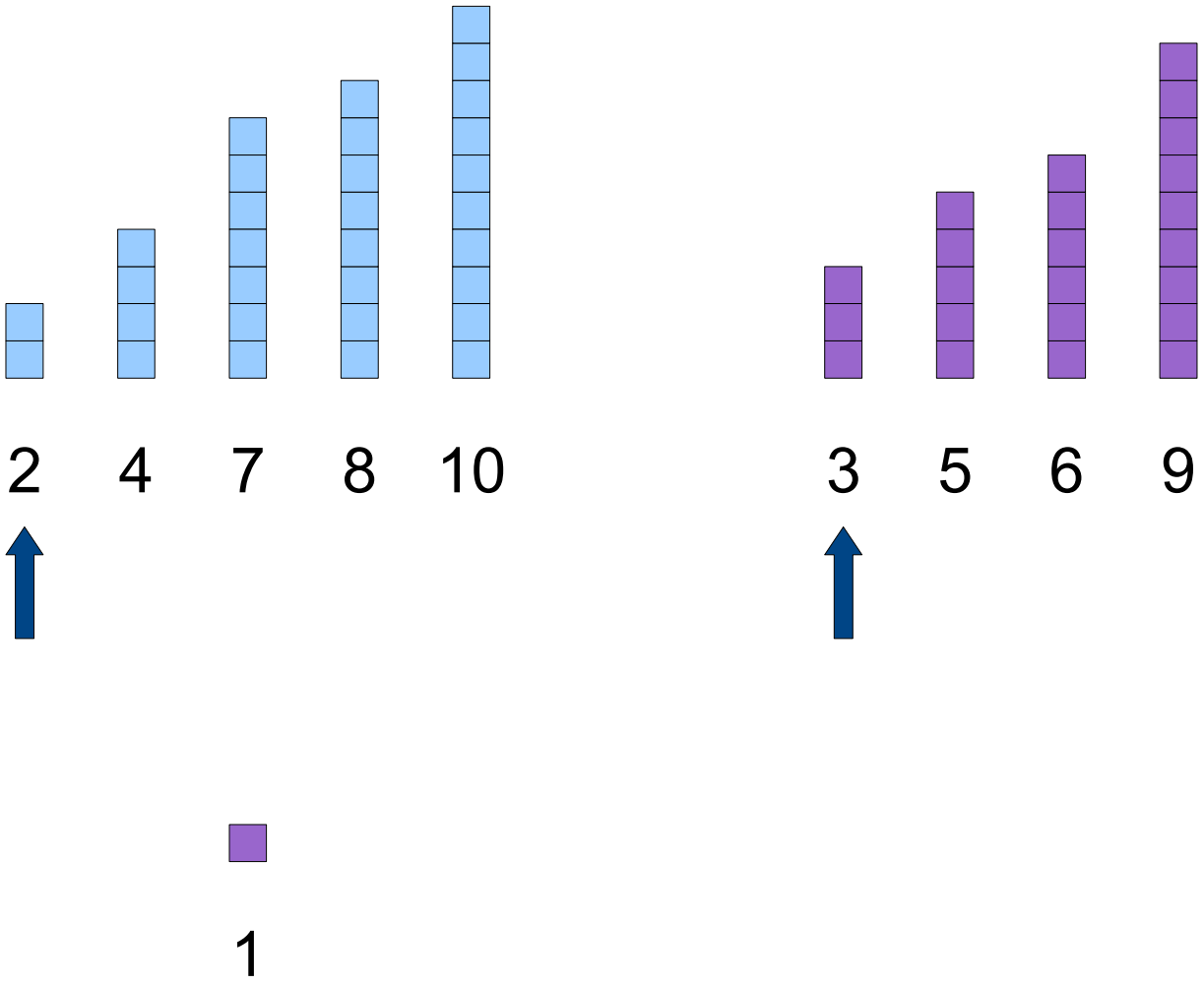
The Key Insight: **Merge**



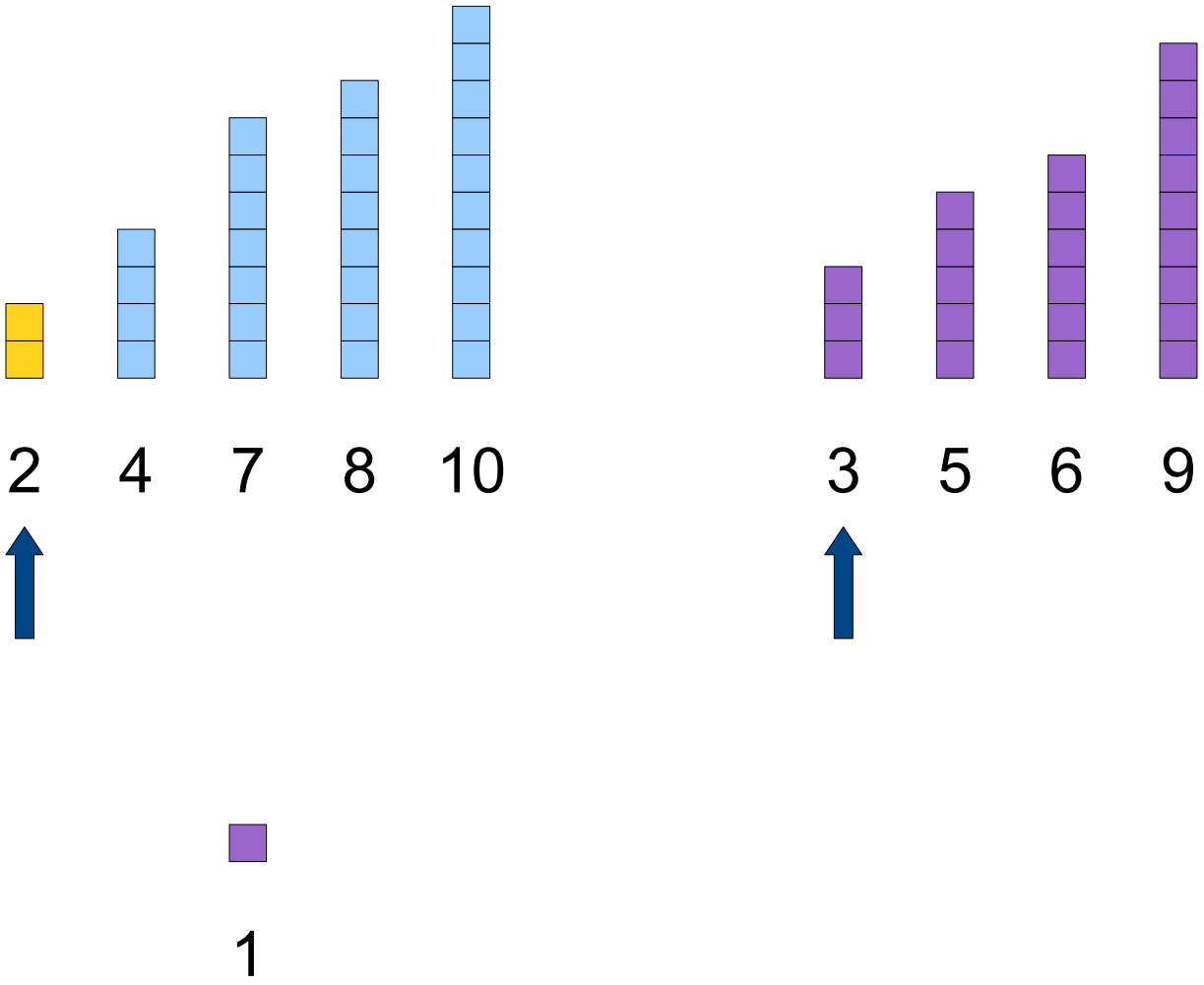
The Key Insight: **Merge**



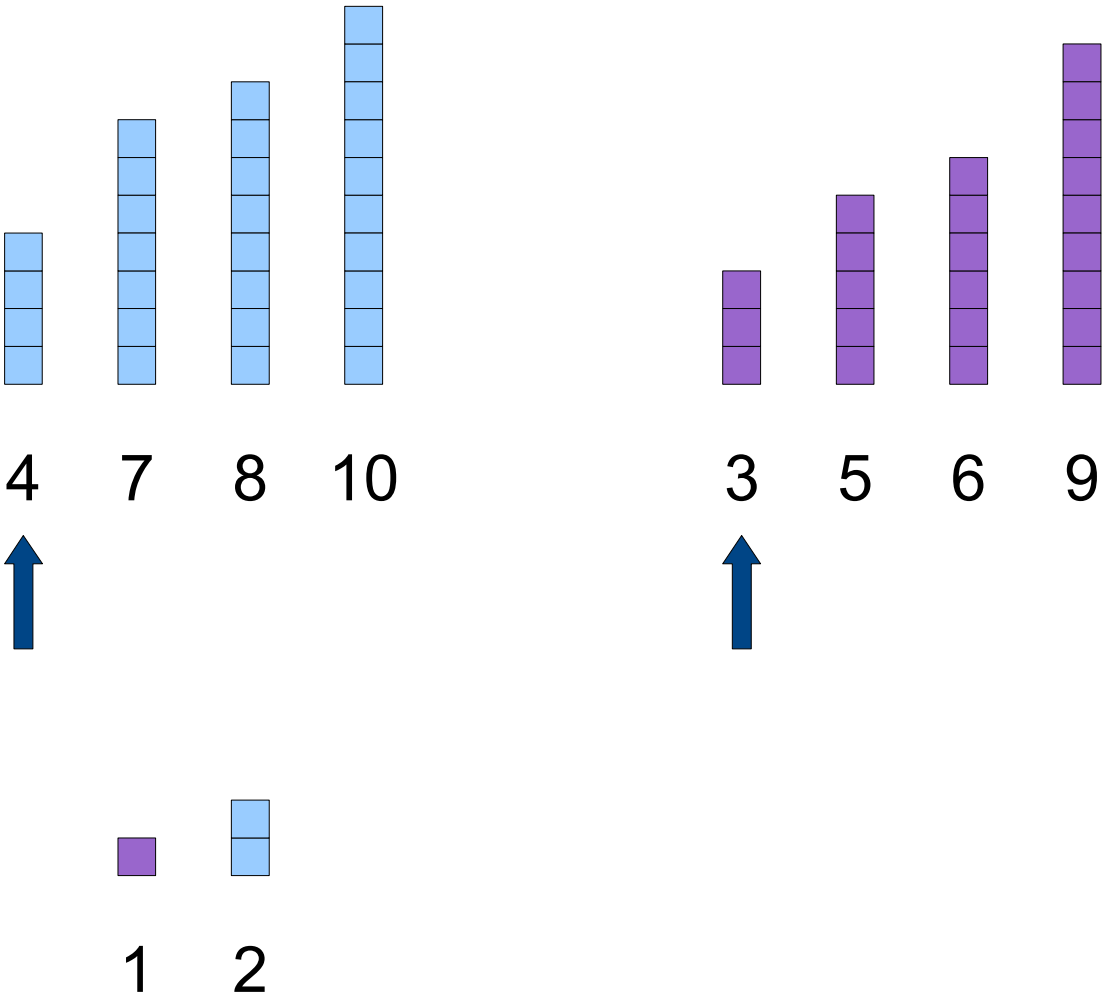
The Key Insight: **Merge**



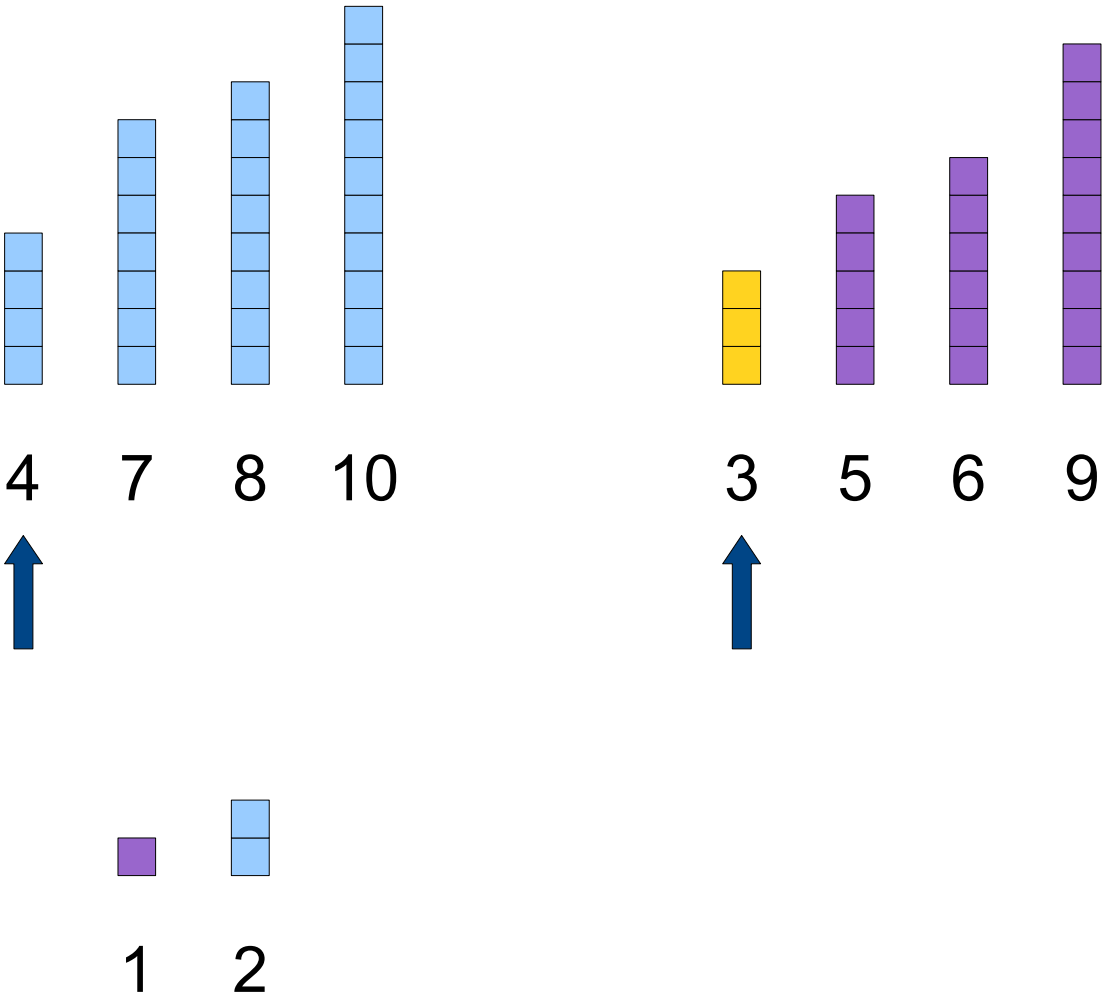
The Key Insight: **Merge**



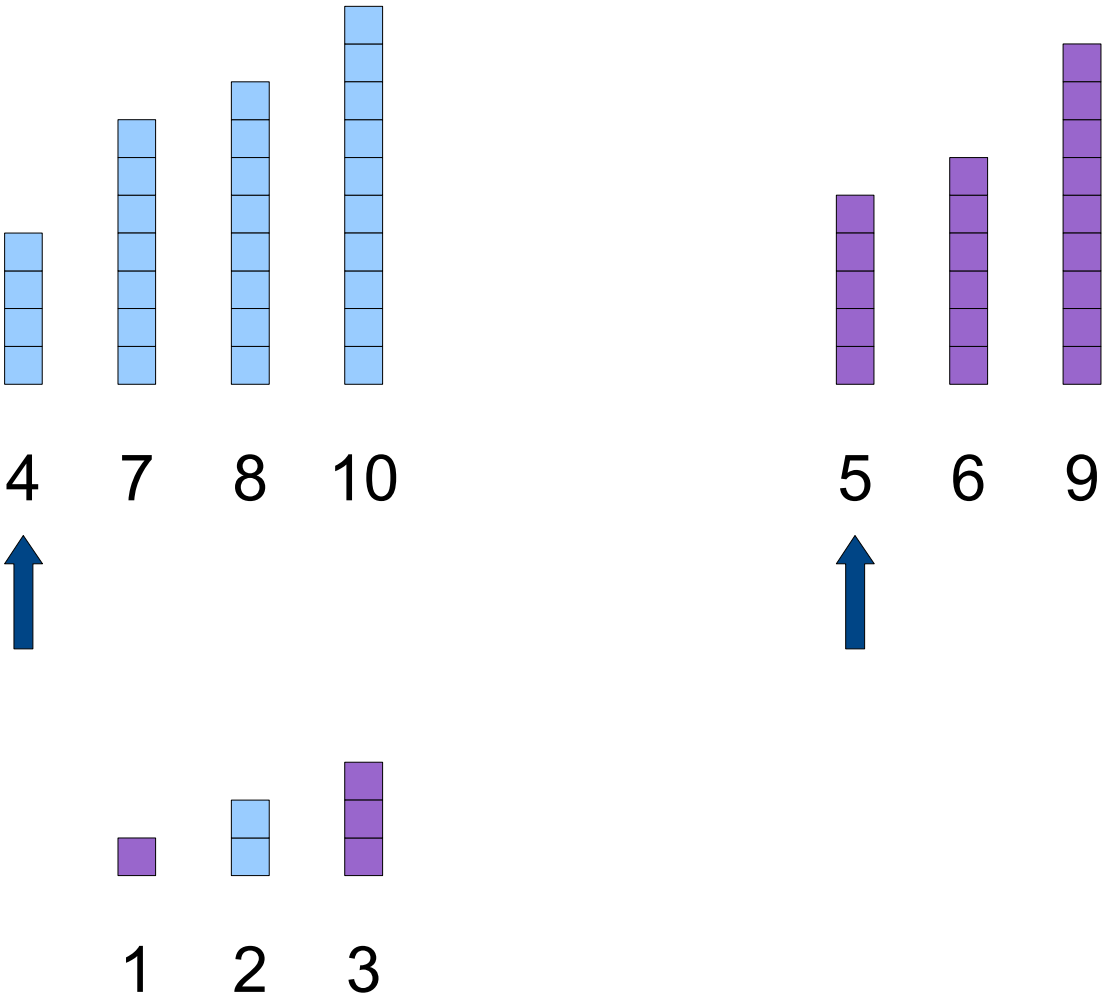
The Key Insight: **Merge**



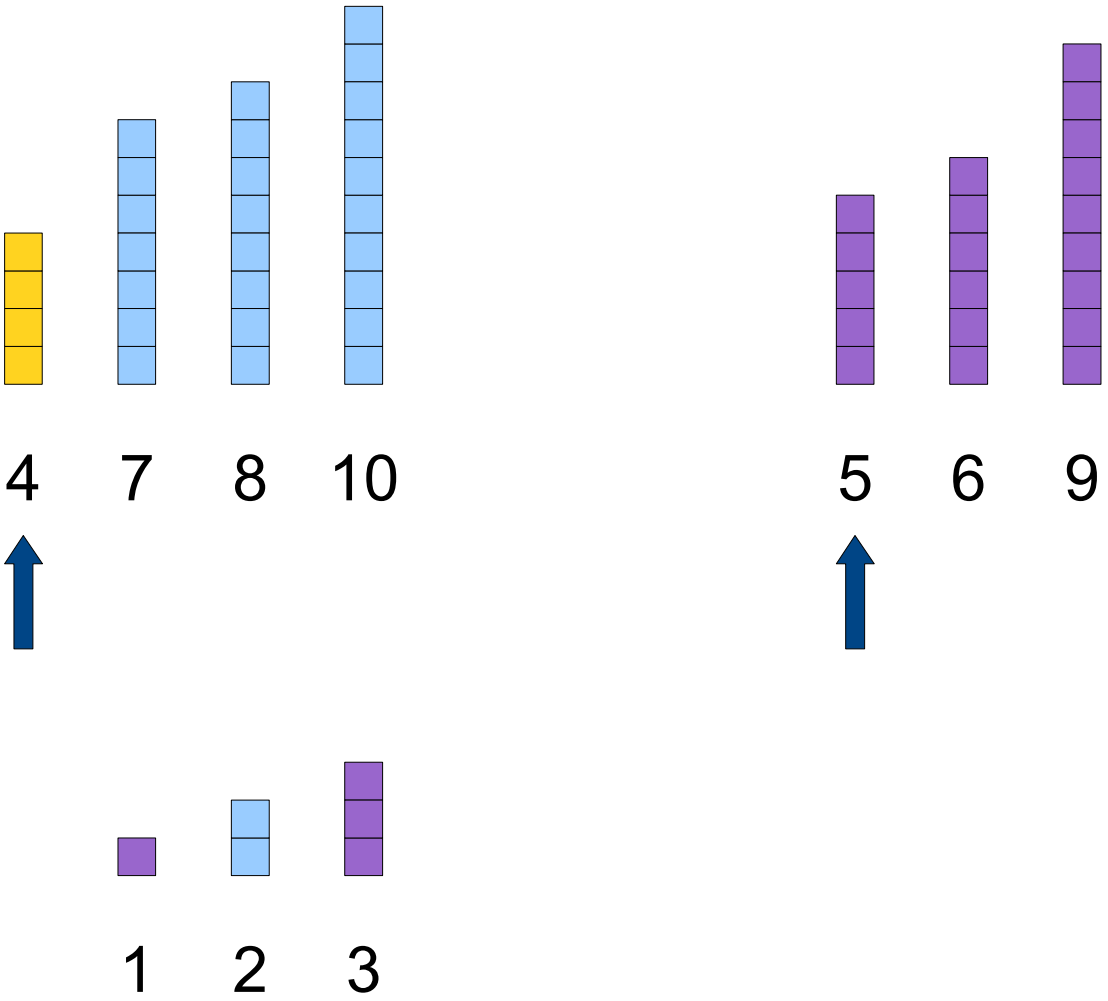
The Key Insight: **Merge**



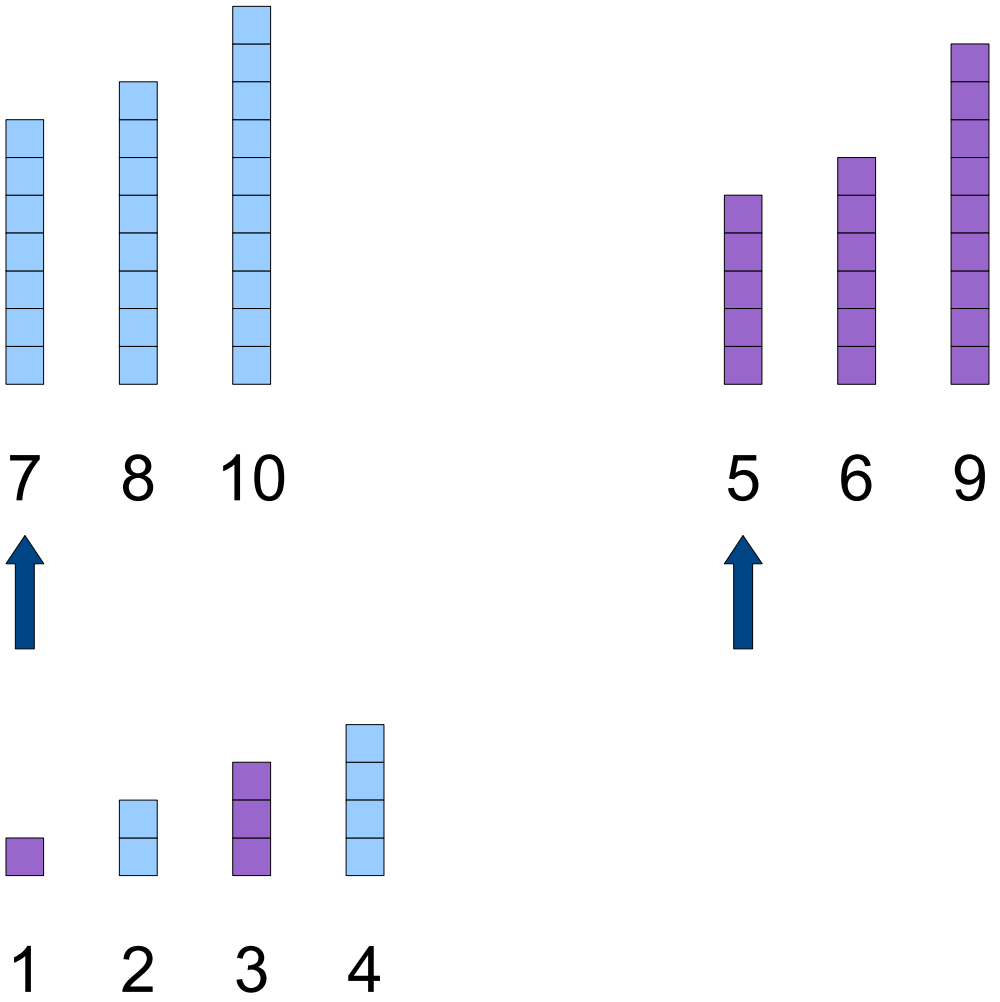
The Key Insight: **Merge**



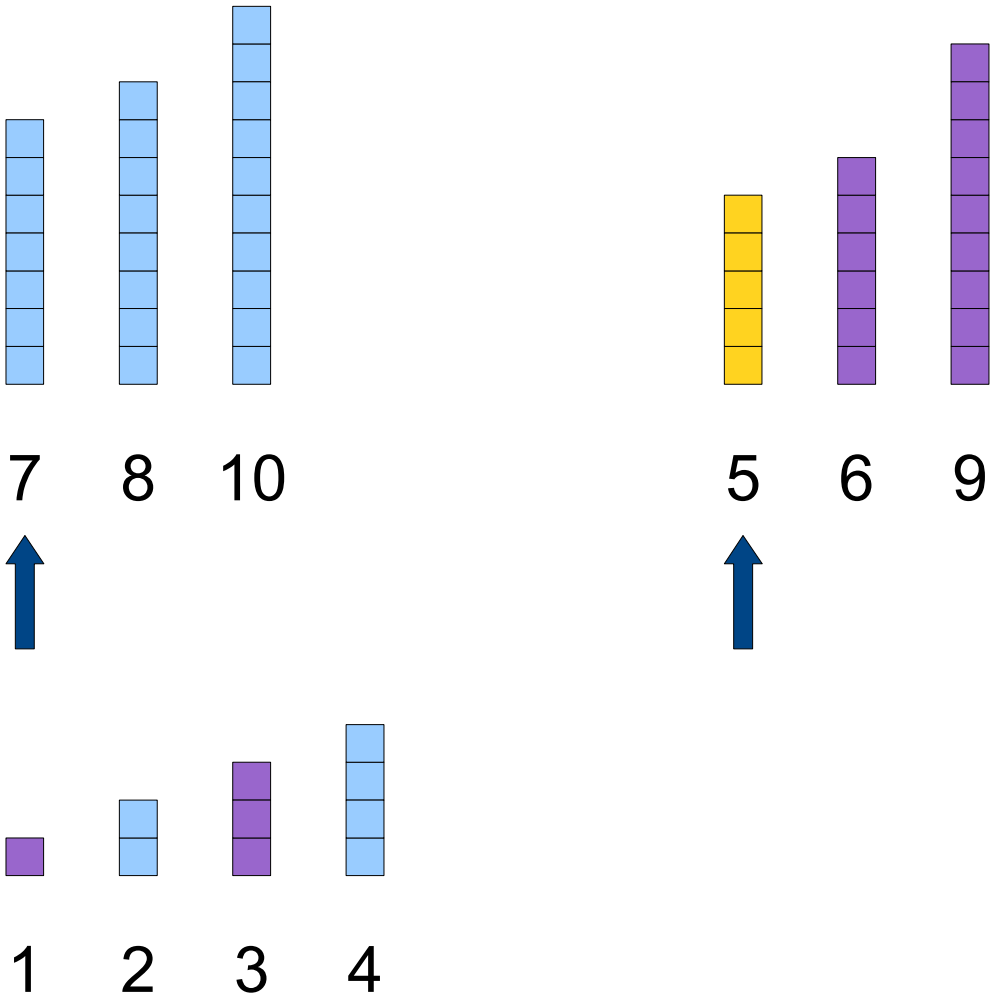
The Key Insight: **Merge**



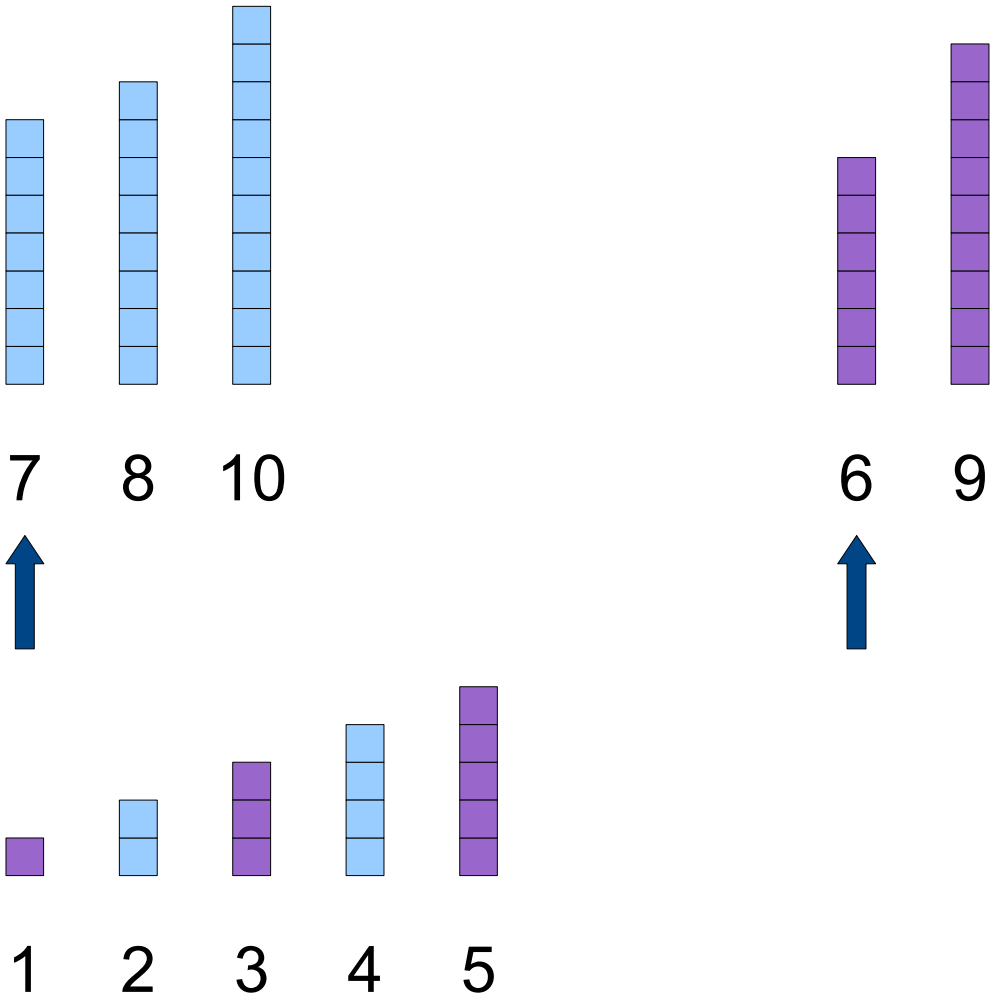
The Key Insight: **Merge**



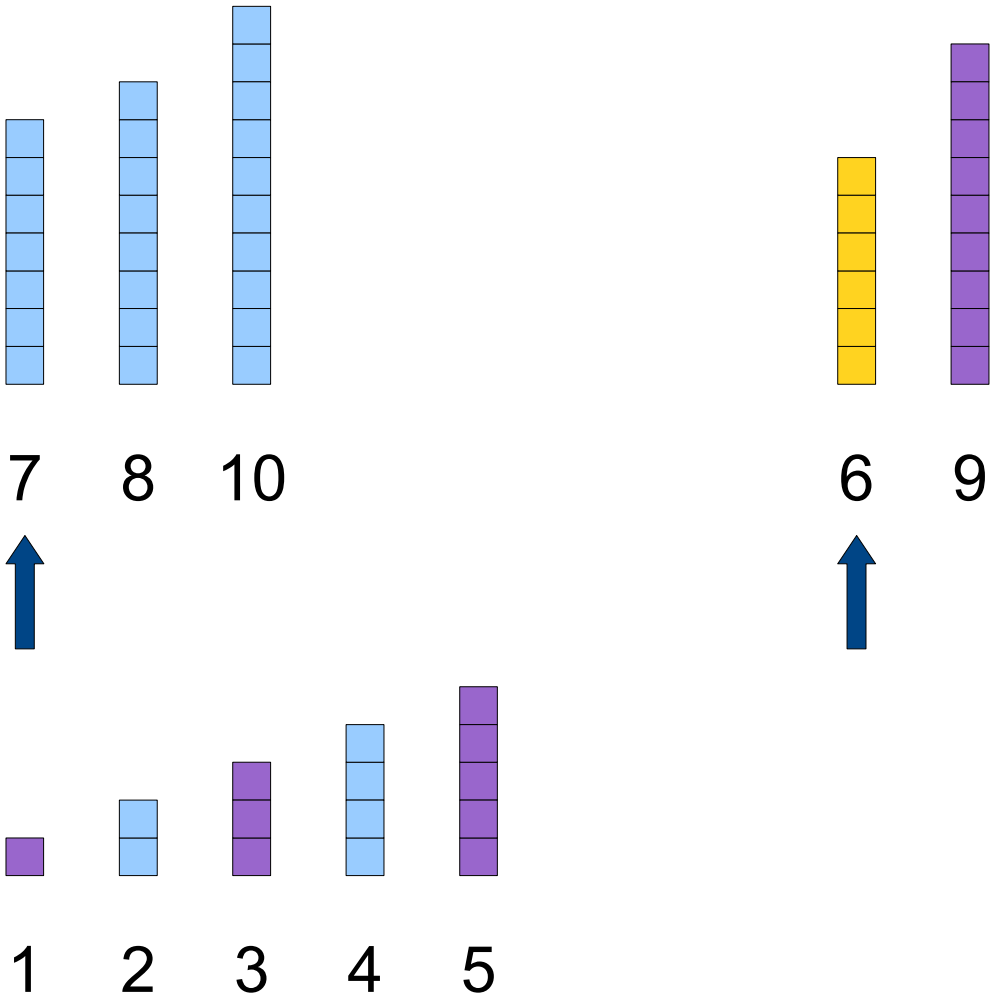
The Key Insight: **Merge**



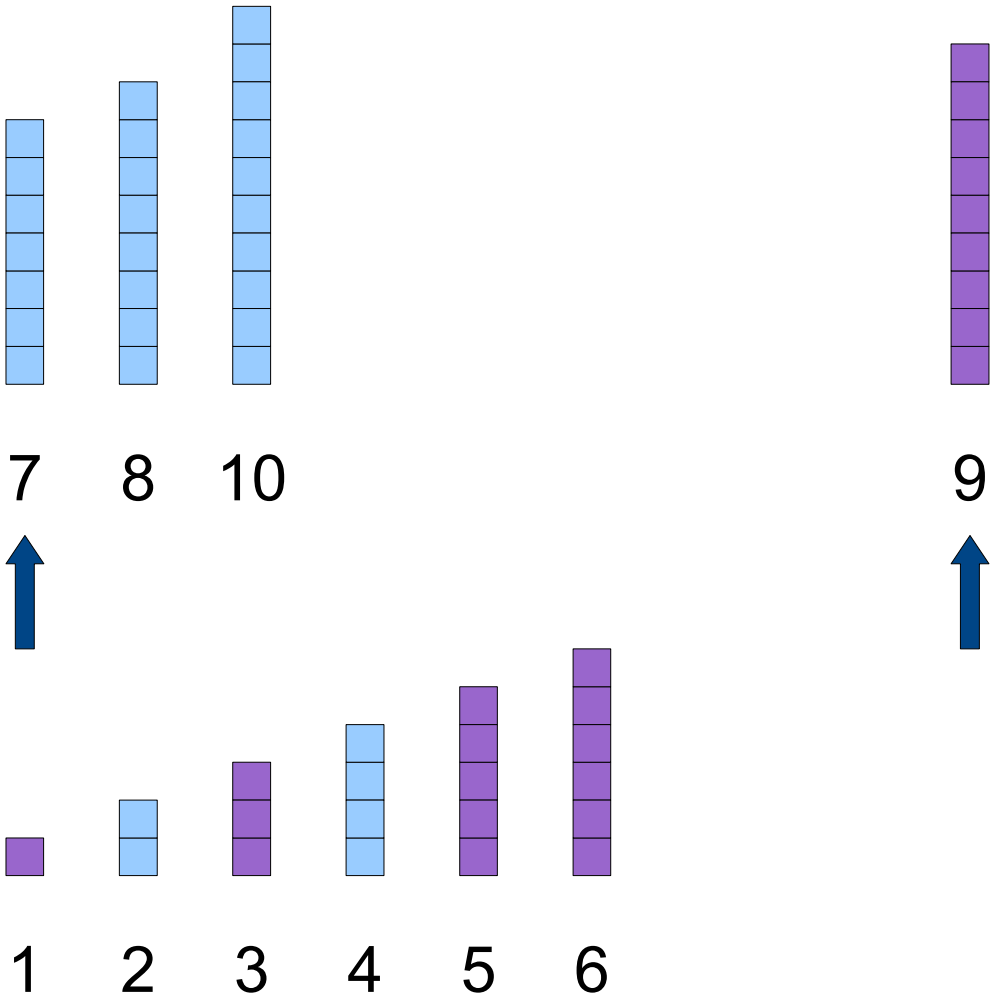
The Key Insight: **Merge**



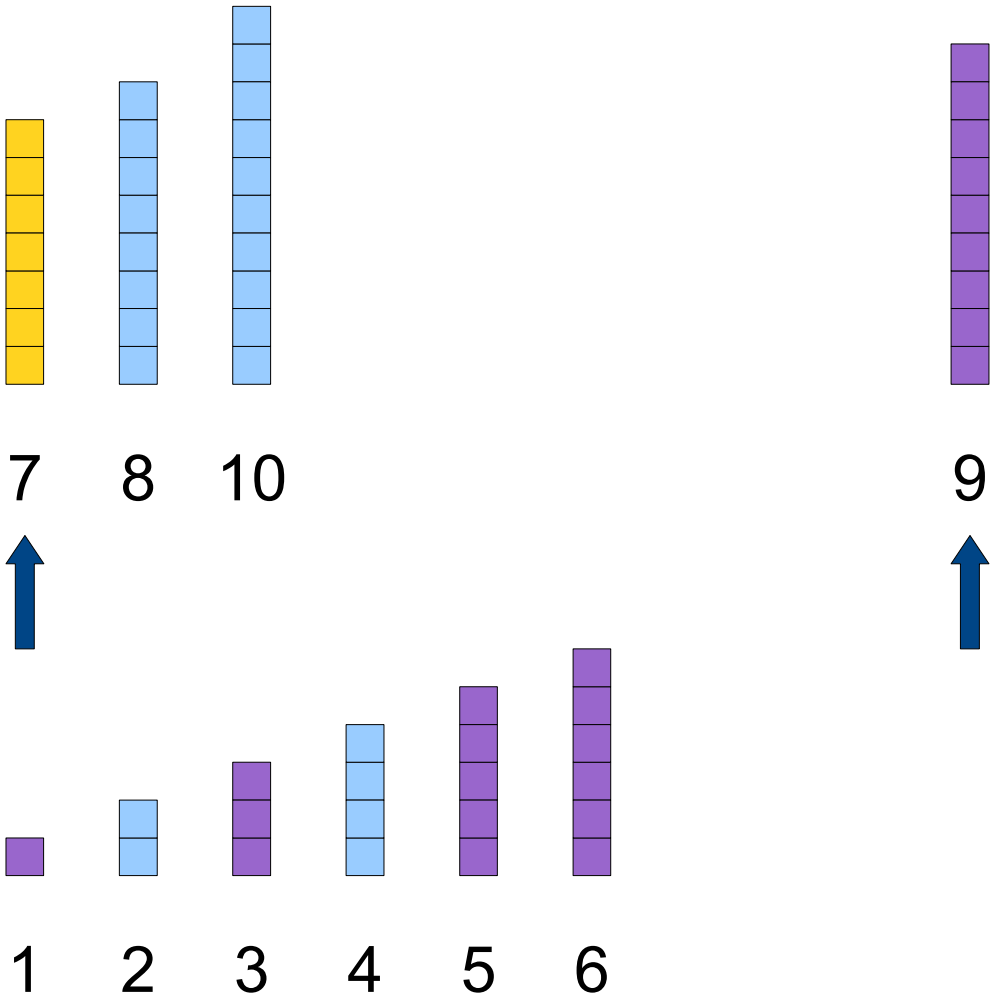
The Key Insight: **Merge**



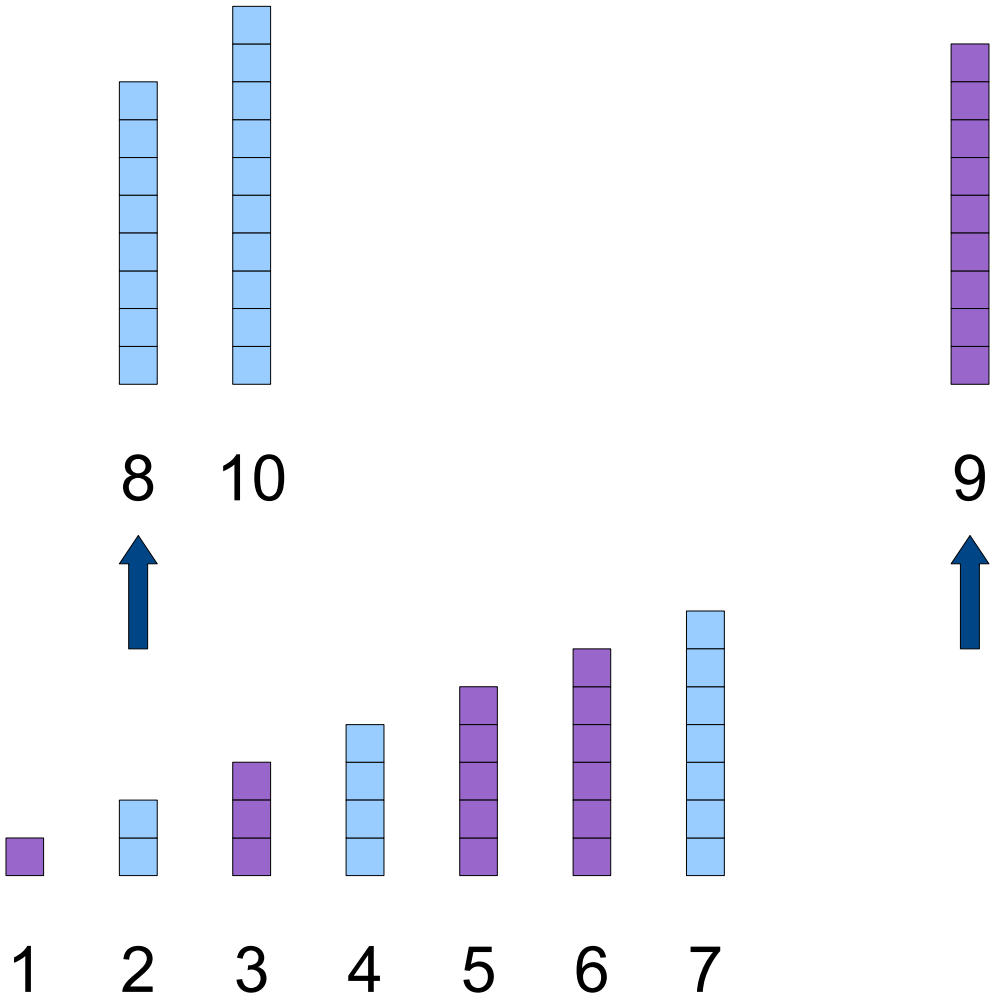
The Key Insight: **Merge**



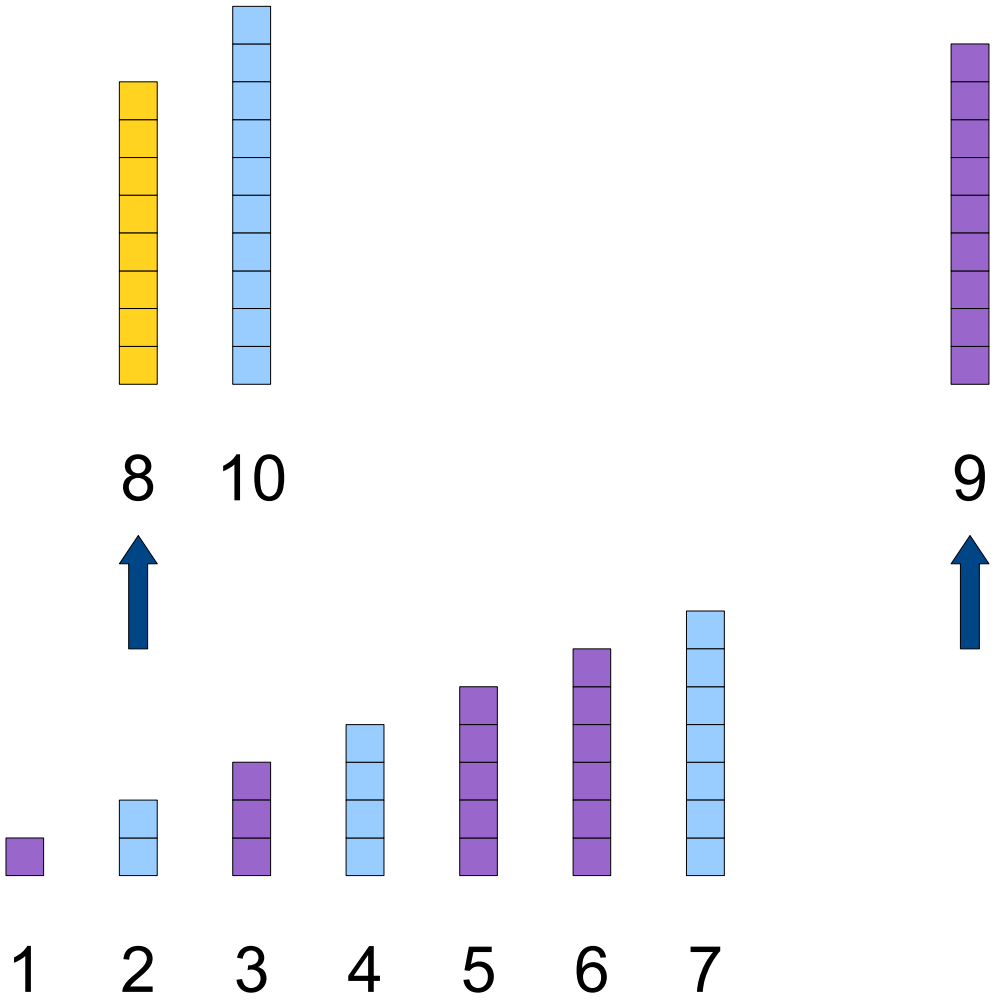
The Key Insight: **Merge**



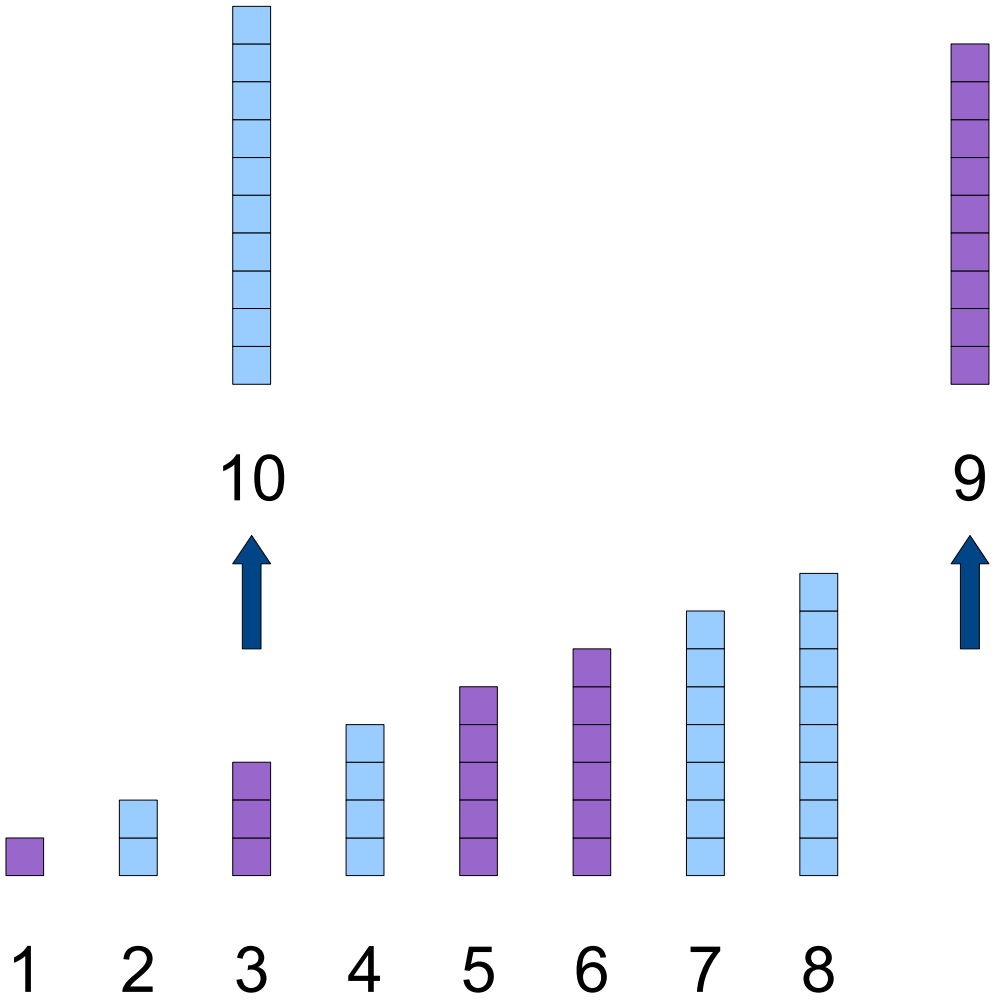
The Key Insight: **Merge**



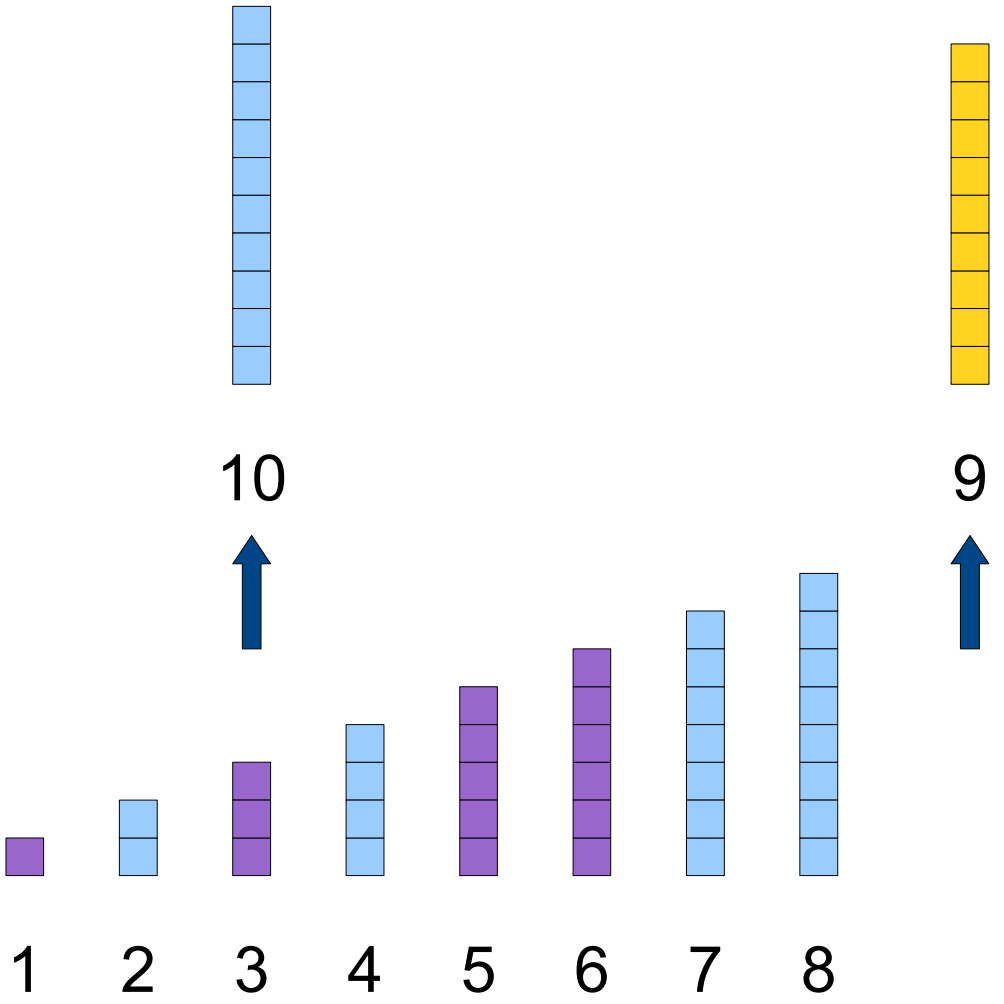
The Key Insight: **Merge**



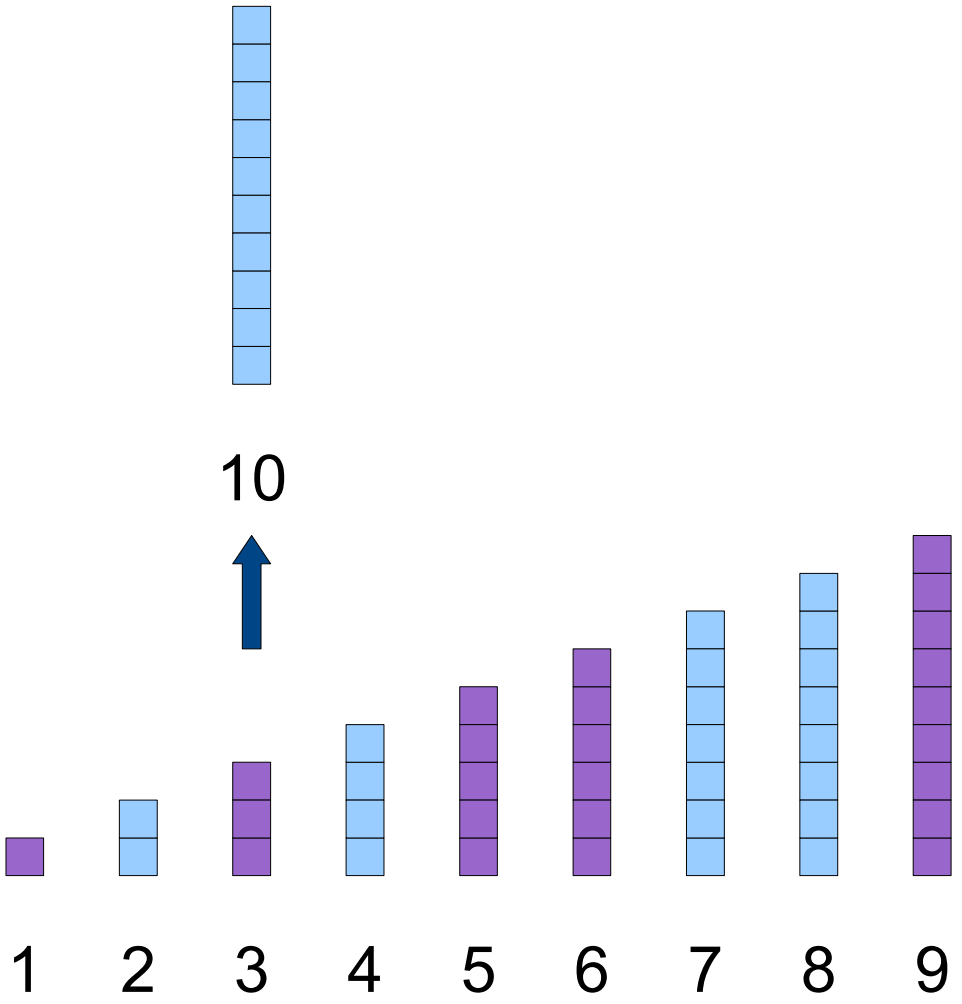
The Key Insight: **Merge**



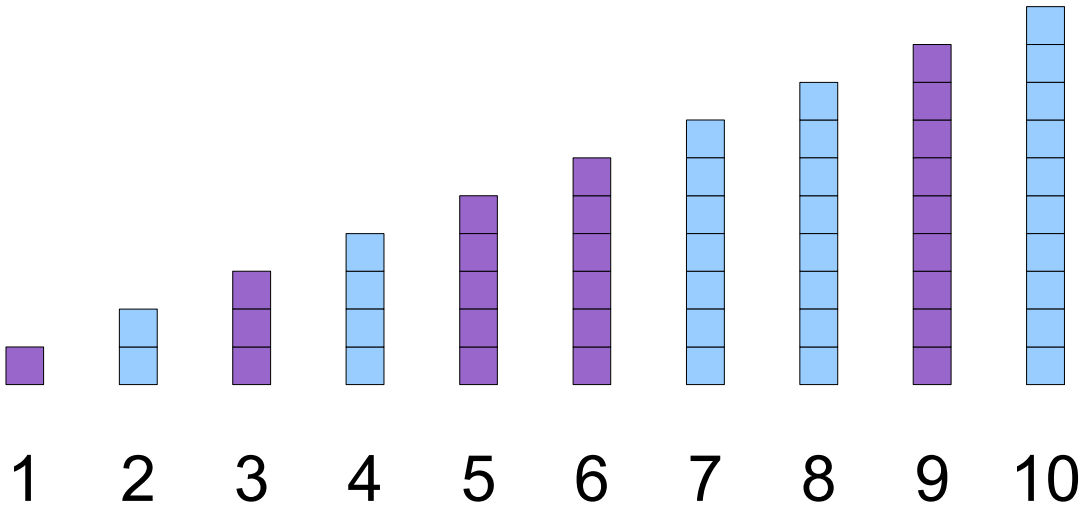
The Key Insight: **Merge**



The Key Insight: **Merge**



The Key Insight: **Merge**



Merge
(Pseudocode)

```
void merge(Queue<int>& one, Queue<int>& two,
           Queue<int>& result) {

  while (!one.isEmpty() && !two.isEmpty()) {
    if (one.peek() < two.peek()) {
      result.enqueue(one.dequeue());
    } else {
      result.enqueue(two.dequeue());
    }
  }

  while (!one.isEmpty()) {
    result.enqueue(one.dequeue());
  }
  while (!two.isEmpty()) {
    result.enqueue(two.dequeue());
  }

  return result;
}
```

“Split Sort”

“Split Sort”

```
void splitSort(Vector<int>& v) {
    Vector<int> left, right;

    for (int i = 0; i < v.size() / 2; i++)
        left += v[i];
    for (int j = v.size() / 2; j < v.size(); j++)
        right += v[i];

    insertionSort(left);
    insertionSort(right);

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

“Split Sort”

```
void splitSort(Vector<int>& v) {
    Vector<int> left, right;

    for (int i = 0; i < v.size() / 2; i++)
        left += v[i];
    for (int j = v.size() / 2; j < v.size(); j++)
        right += v[i];

    insertionSort(left);
    insertionSort(right);

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

What is the Big-O runtime of this function?

“Split Sort”

```
void splitSort(Vector<int>& v) {
    Vector<int> left, right;

    for (int i = 0; i < v.size() / 2; i++)
        left += v[i];
    for (int j = v.size() / 2; j < v.size(); j++)
        right += v[i];

    insertionSort(left);
    insertionSort(right);

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

What is the Big-O runtime of this function?

$O(n^2)$

Performance Comparison

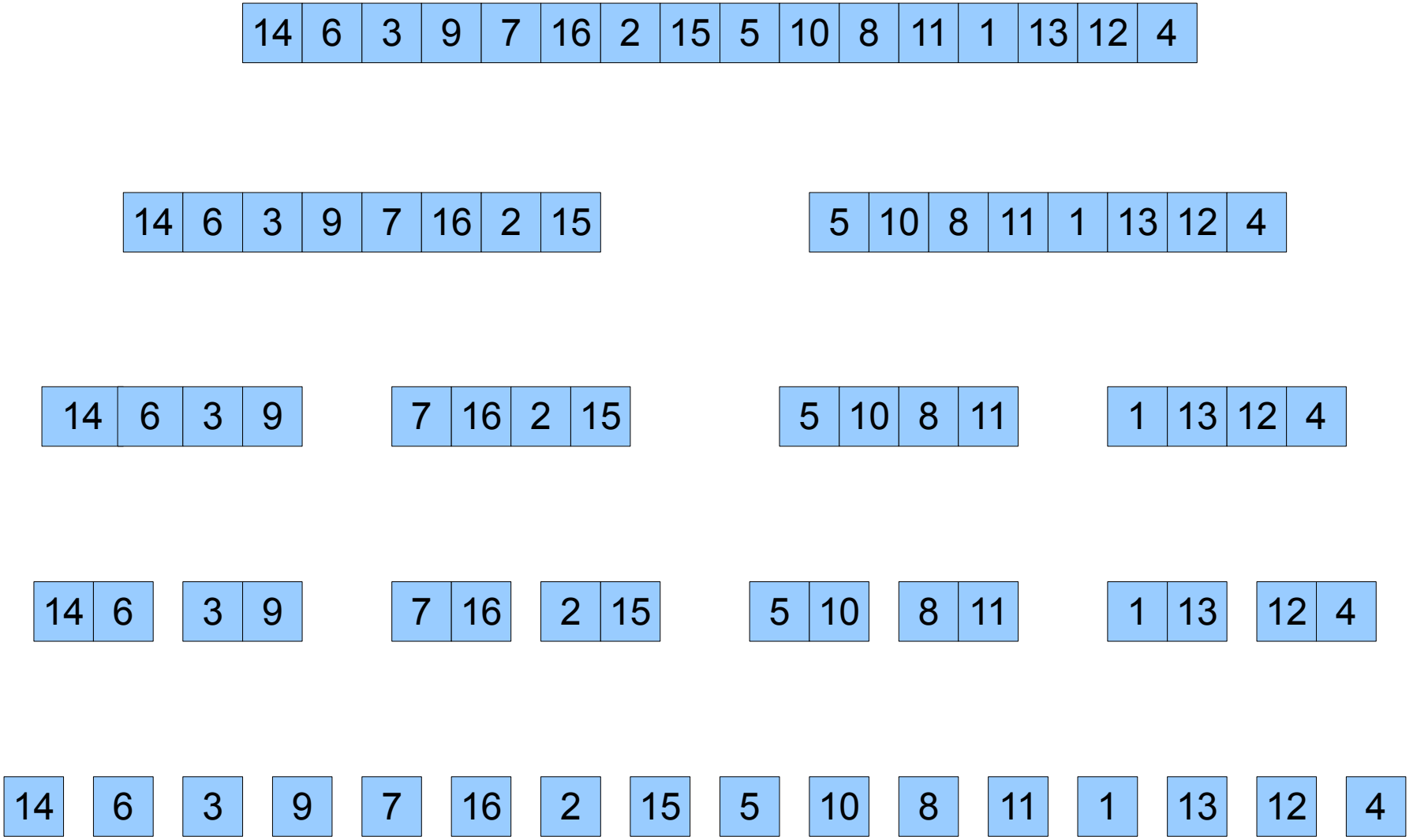
Size	Selection Sort	Insertion Sort
10000	0.304	0.160
20000	1.218	0.630
30000	2.790	1.427
40000	4.646	2.520
50000	7.395	4.181
60000	10.584	5.635
70000	14.149	8.143
80000	18.674	10.333
90000	23.165	12.832

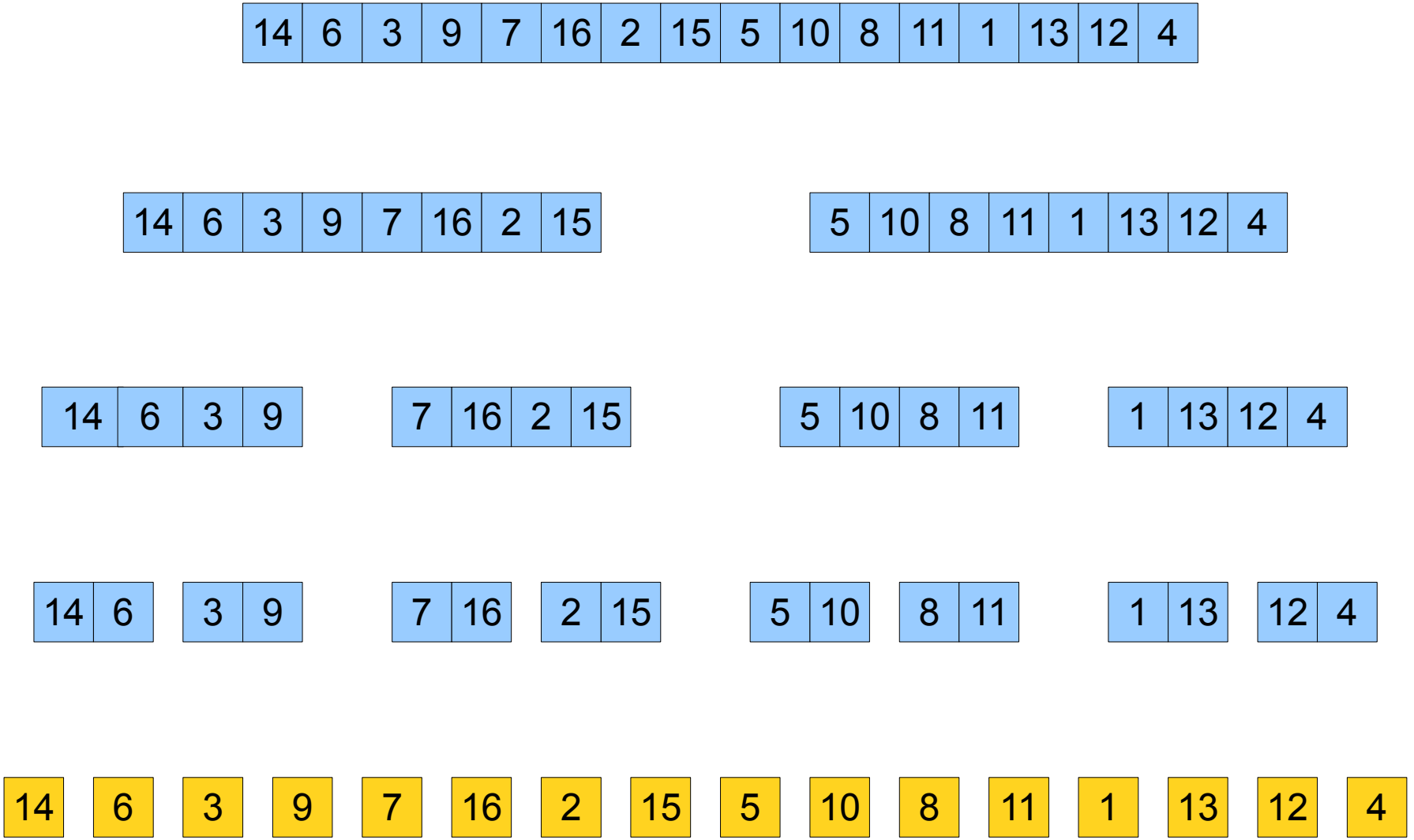
Performance Comparison

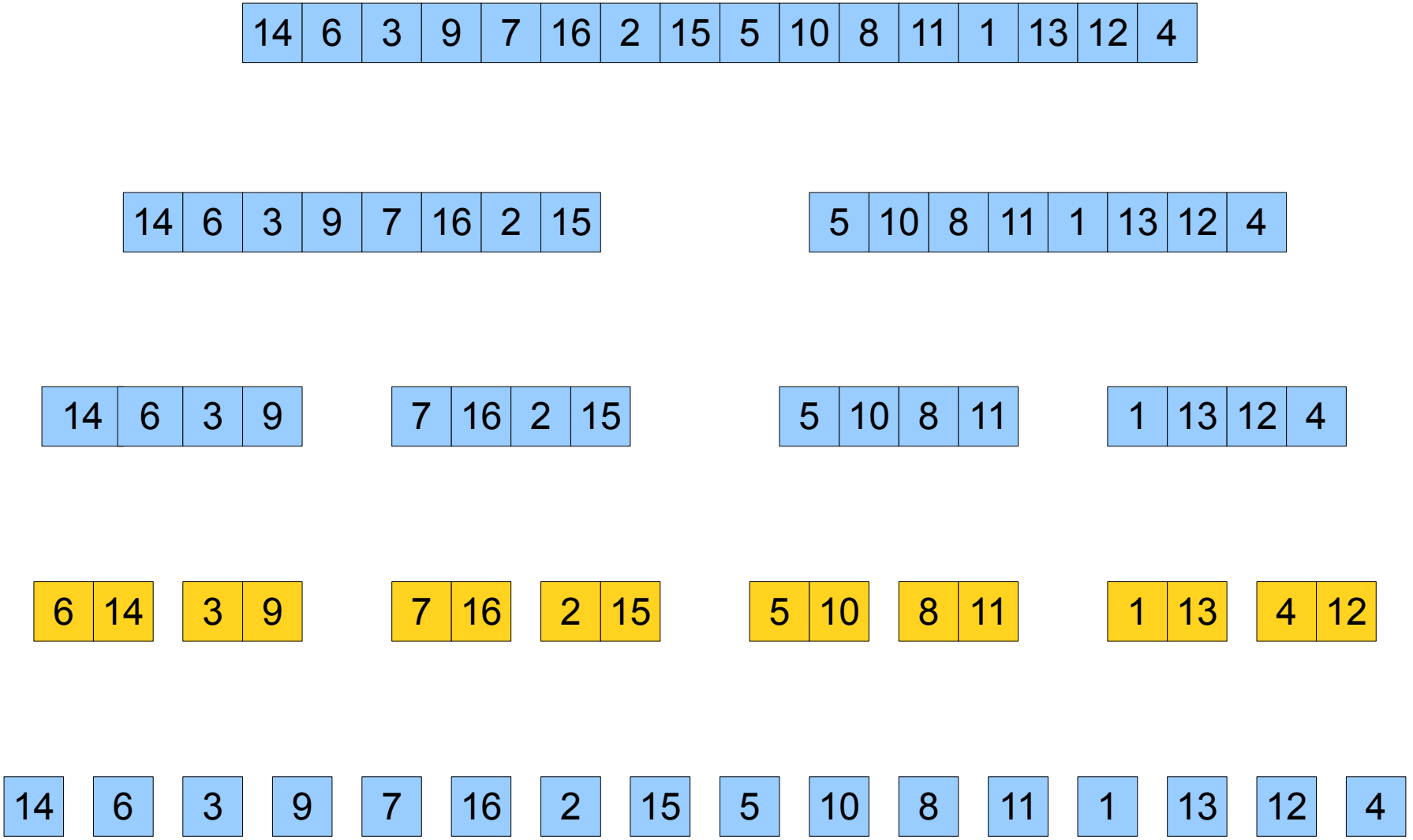
Size	Selection Sort	Insertion Sort	"Split Sort"
10000	0.304	0.160	0.161
20000	1.218	0.630	0.387
30000	2.790	1.427	0.726
40000	4.646	2.520	1.285
50000	7.395	4.181	2.719
60000	10.584	5.635	2.897
70000	14.149	8.143	3.939
80000	18.674	10.333	5.079
90000	23.165	12.832	6.375

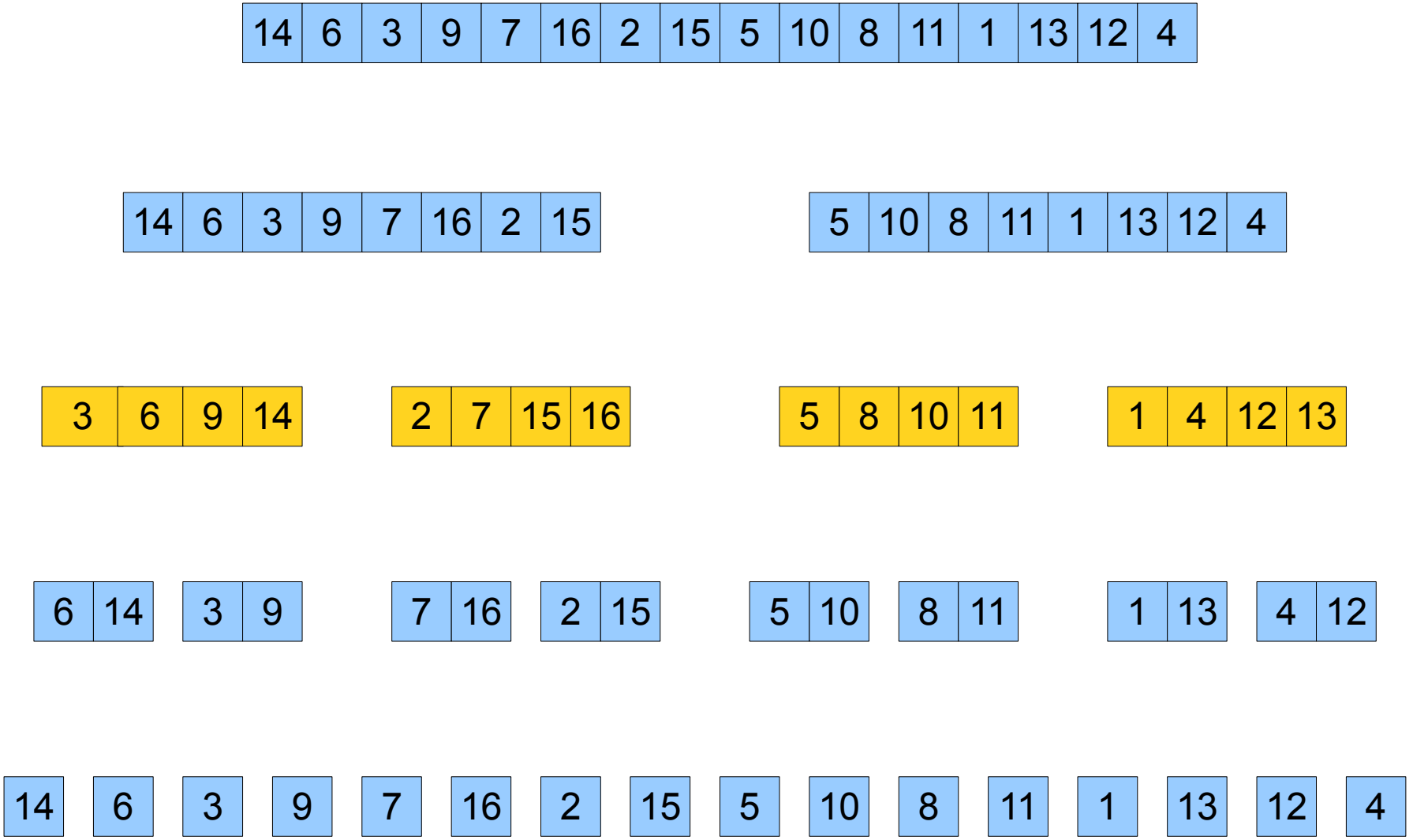
A Better Idea

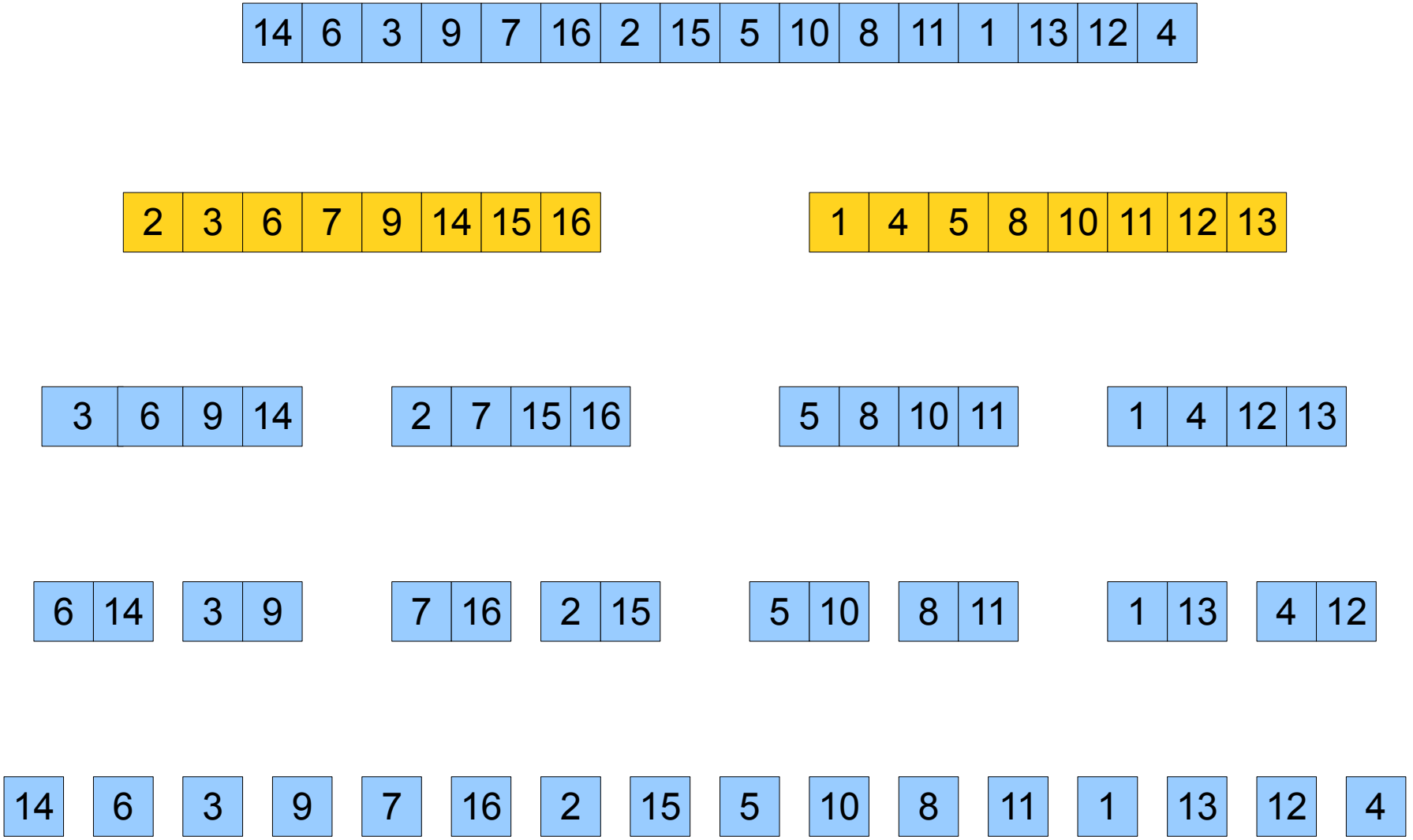
- Splitting the input in half and merging halves the work.
- So why not split into four? Or eight?
- **Question**: What happens if we *never stop splitting?*

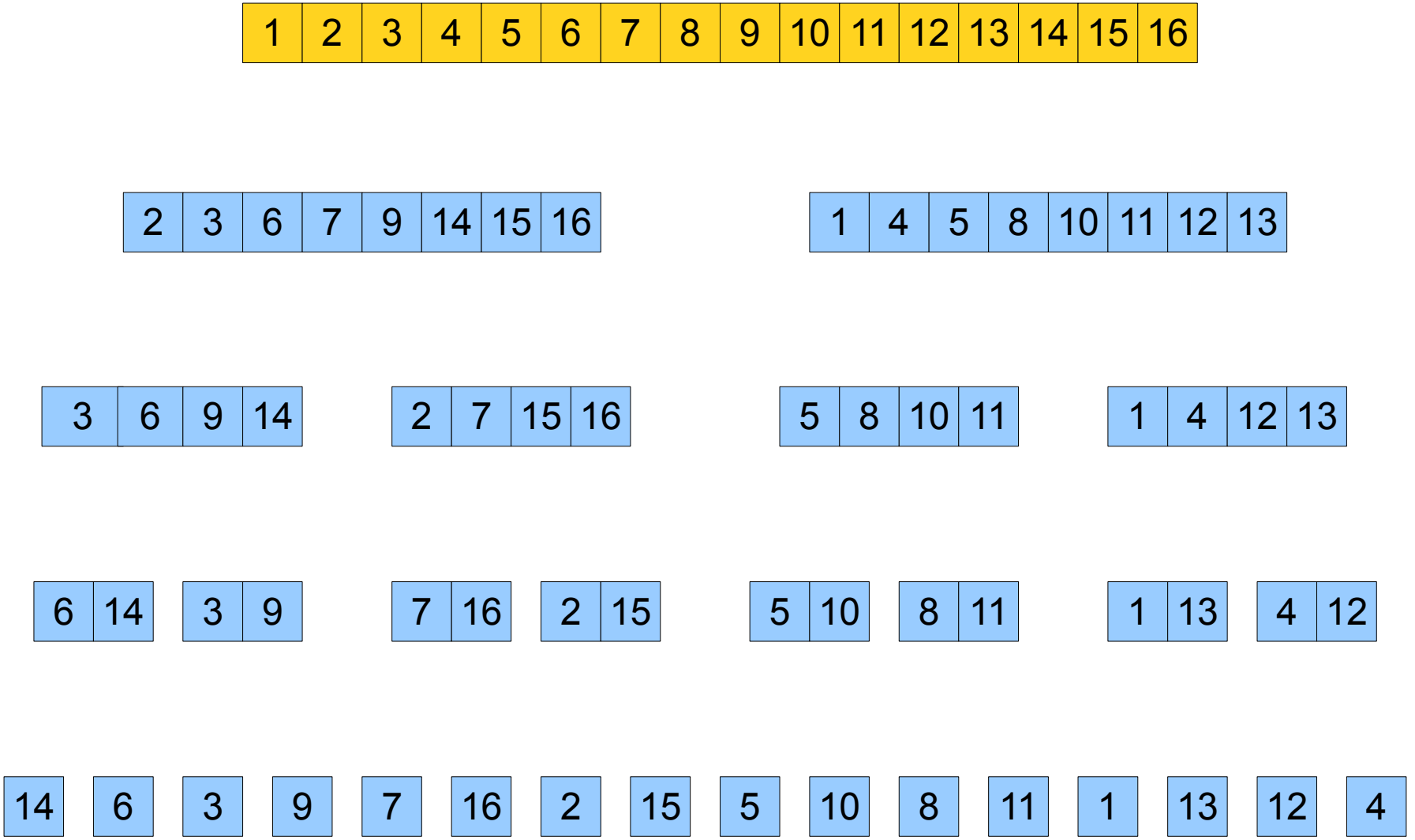

14 6 3 9 7 16 2 15 5 10 8 11 1 13 12 4
14 6 3 9 7 16 2 15
5 10 8 11 1 13 12 4
14 6 3 9
7 16 2 15
5 10 8 11
1 13 12 4
14 6
3 9
7 16
2 15
5 10
8 11
1 13
12 4
14
6
3
9
7
16
2
15
5
10
8
11
1
13
12
4


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


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


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


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


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

MergeSort (Pseudocode)

High-Level Idea

- A recursive sorting algorithm!
- **Base Case**:
  - An empty or single-element list is already sorted.
- **Recursive step**:
  - Break the list in half and recursively sort each part.
  - Use `merge` to combine them back into a single sorted list.
- This algorithm is called *mergesort*.

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

    /* Split v into two subvectors. */
    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++)
        left += v[i];
    for (int i = v.size() / 2; i < v.size(); i++)
        right += v[i];

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

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

What is the complexity of mergesort?

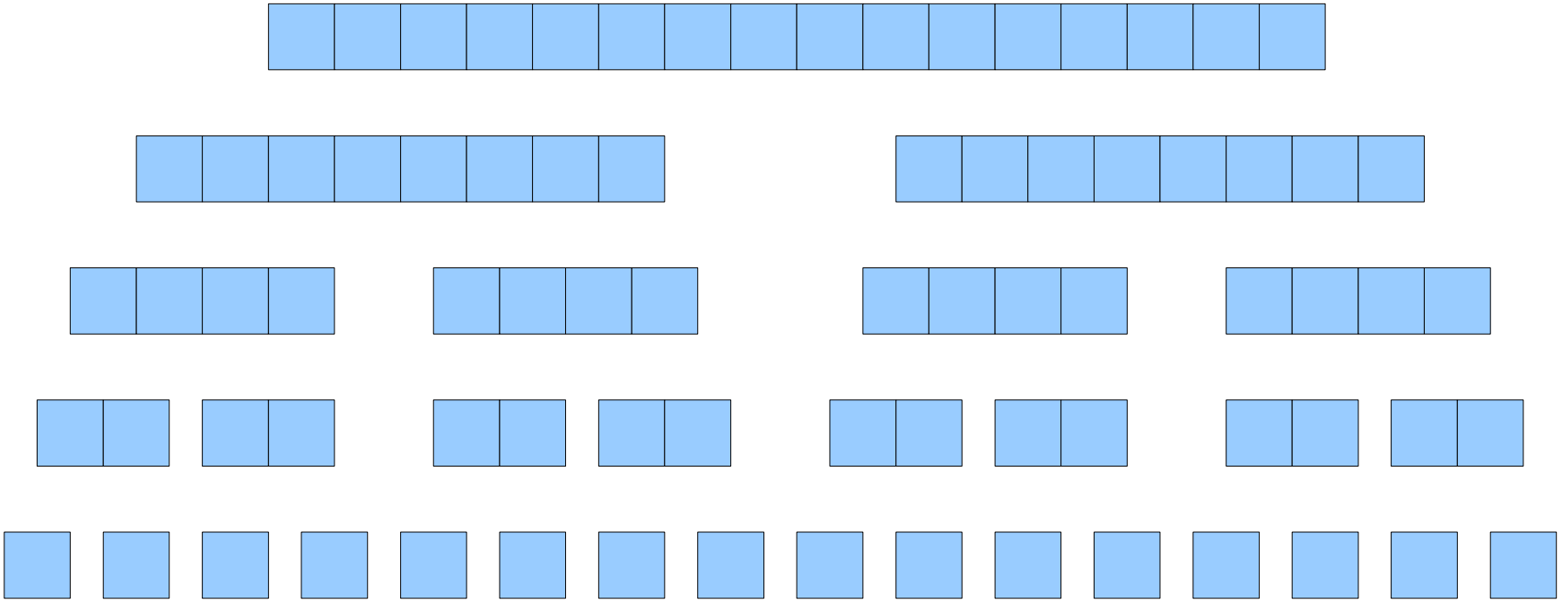
```
void mergesort(Vector<int>& v) {
   /* Base case: 0- or 1-element lists are
    * already sorted.
    */
   if (v.size() <= 1) return;

   /* Split v into two subvectors. */
   Vector<int> left, right;
   for (int i = 0; i < v.size() / 2; i++)
       left += v[i];
   for (int i = v.size() / 2; i < v.size(); i++)
       right += v[i];

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

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

A Graphical Intuition



O(*n*)

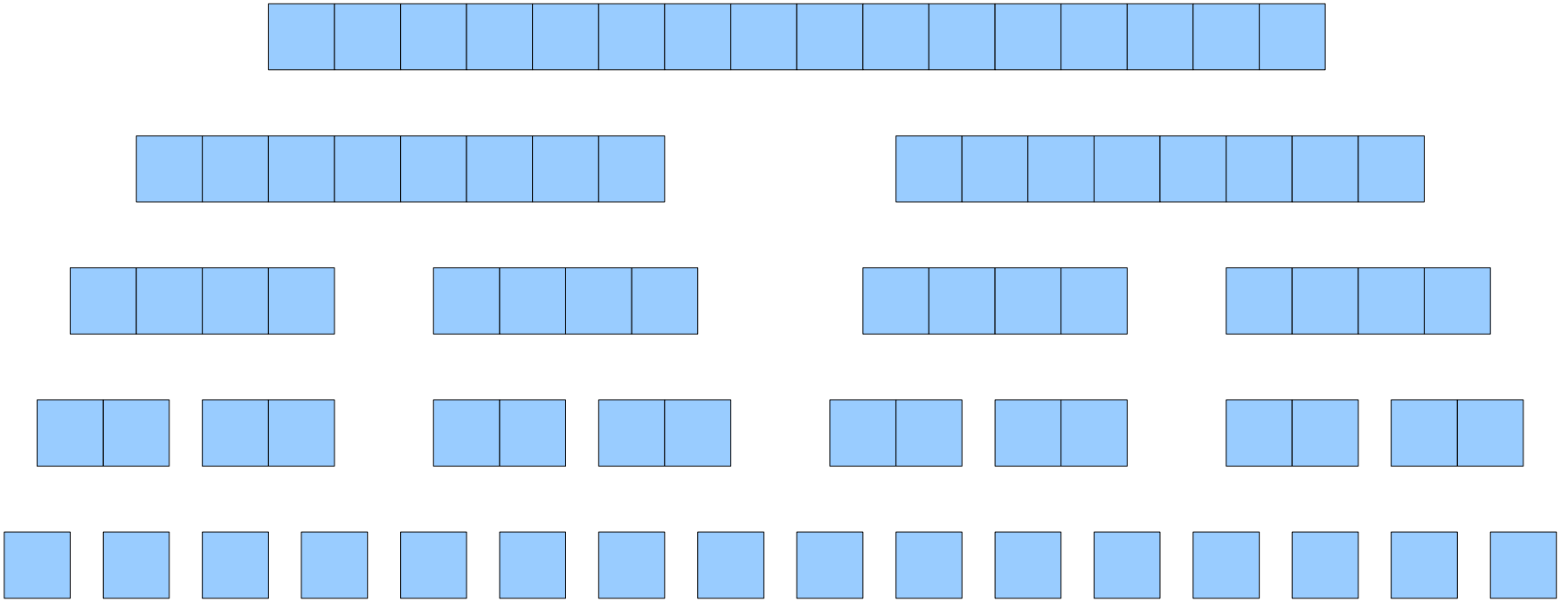
O(*n*)

O(*n*)

O(*n*)

O(*n*)

A Graphical Intuition



How many levels are there?

Slicing and Dicing

- After zero recursive calls: n
- After one recursive call: $n / 2$
- After two recursive calls: $n / 4$
- After three recursive calls: $n / 8$

...

- After k recursive calls: $n / 2^k$

Cutting in Half

- After k recursive calls, there are n / 2^k elements left.
- Mergesort stops recursing when there are zero or one elements left.
- Solving for the number of levels:

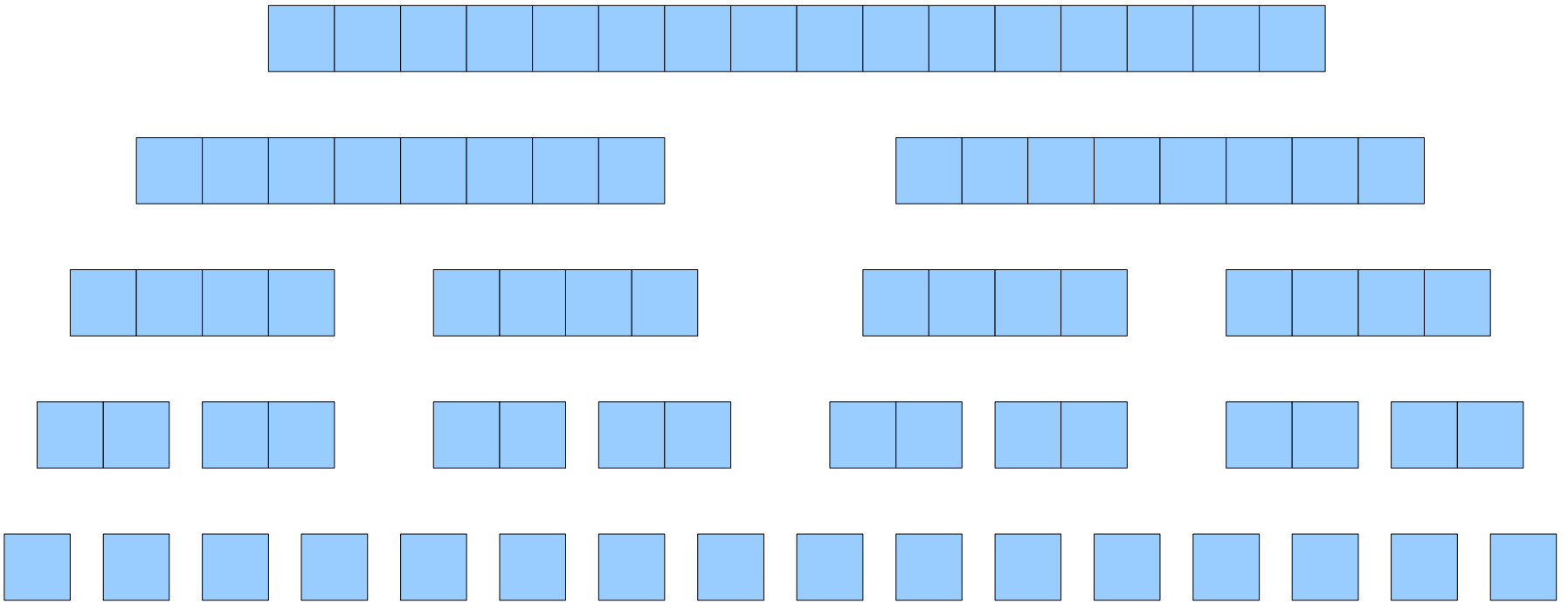
$$n / 2^k = 1$$

$$n = 2^k$$

$$\log_2 n = k$$

- So mergesort recurses **$\log_2 n$** levels deep.

A Graphical Intuition



O(*n* log *n*)

Mergesort Times

Size	Selection Sort	Insertion Sort	“Split Sort”
10000	0.304	0.160	0.161
20000	1.218	0.630	0.387
30000	2.790	1.427	0.726
40000	4.646	2.520	1.285
50000	7.395	4.181	2.719
60000	10.584	5.635	2.897
70000	14.149	8.143	3.939
80000	18.674	10.333	5.079
90000	23.165	12.832	6.375

Mergesort Times

Size	Selection Sort	Insertion Sort	"Split Sort"	Mergesort
10000	0.304	0.160	0.161	0.006
20000	1.218	0.630	0.387	0.010
30000	2.790	1.427	0.726	0.017
40000	4.646	2.520	1.285	0.021
50000	7.395	4.181	2.719	0.028
60000	10.584	5.635	2.897	0.035
70000	14.149	8.143	3.939	0.041
80000	18.674	10.333	5.079	0.042
90000	23.165	12.832	6.375	0.048

Can we do Better?

- Mergesort is O(n log n).
- This is asymptotically better than O(n^2)
- Can we do better?
  - In general, **no**: comparison-based sorts cannot have a worst-case runtime better than O(n log n).
- **In the worst case, we can only get faster by a constant factor!**

Growth Rates