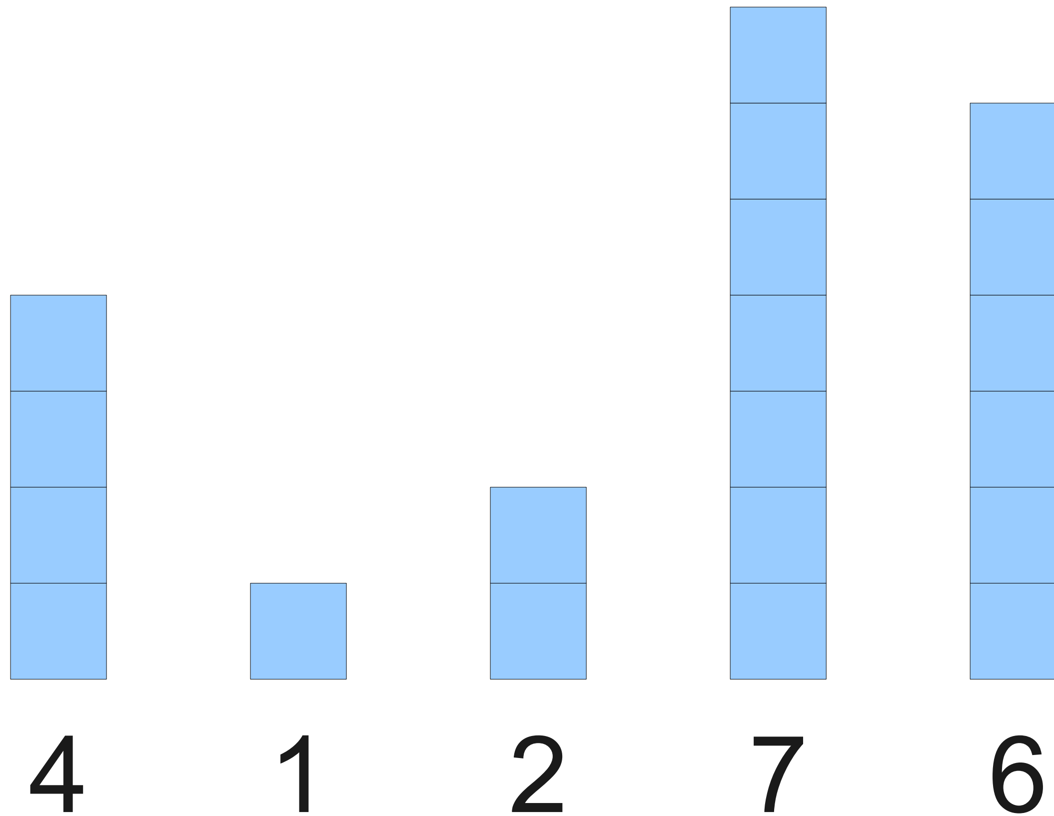
Algorithmic Analysis and Sorting, Part Two

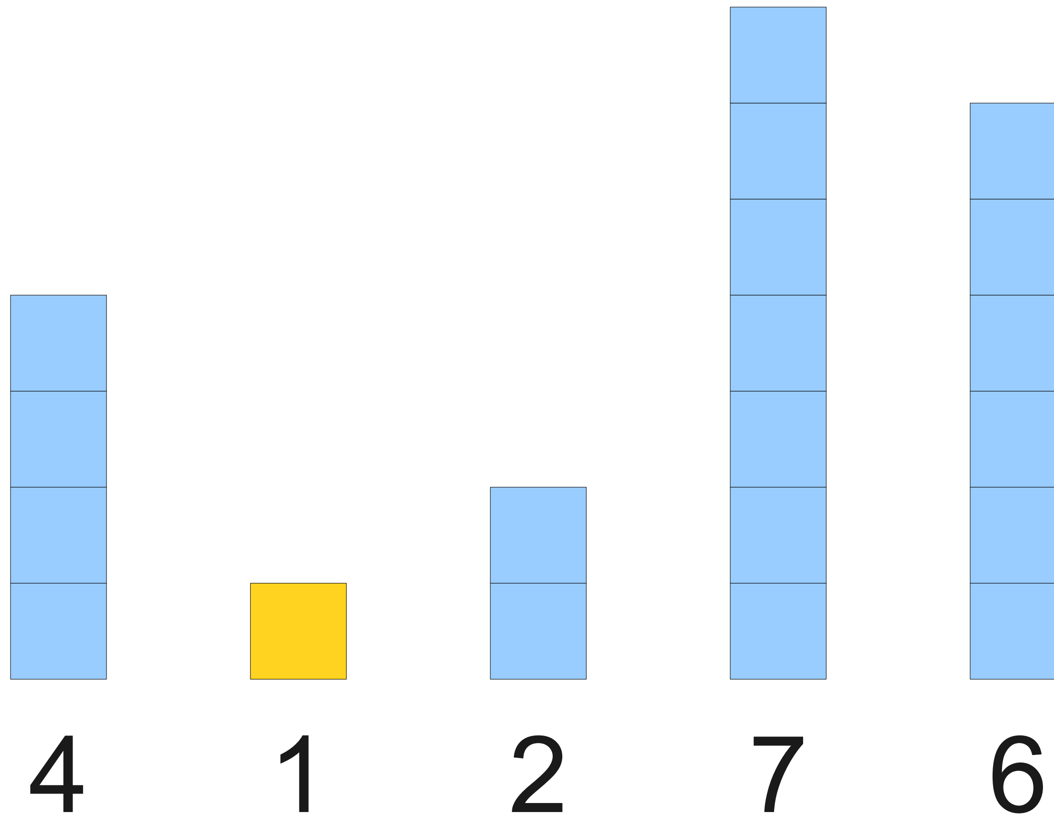
Friday Four Square!
4:15PM, Outside Gates

An Initial Idea: **Selection Sort**

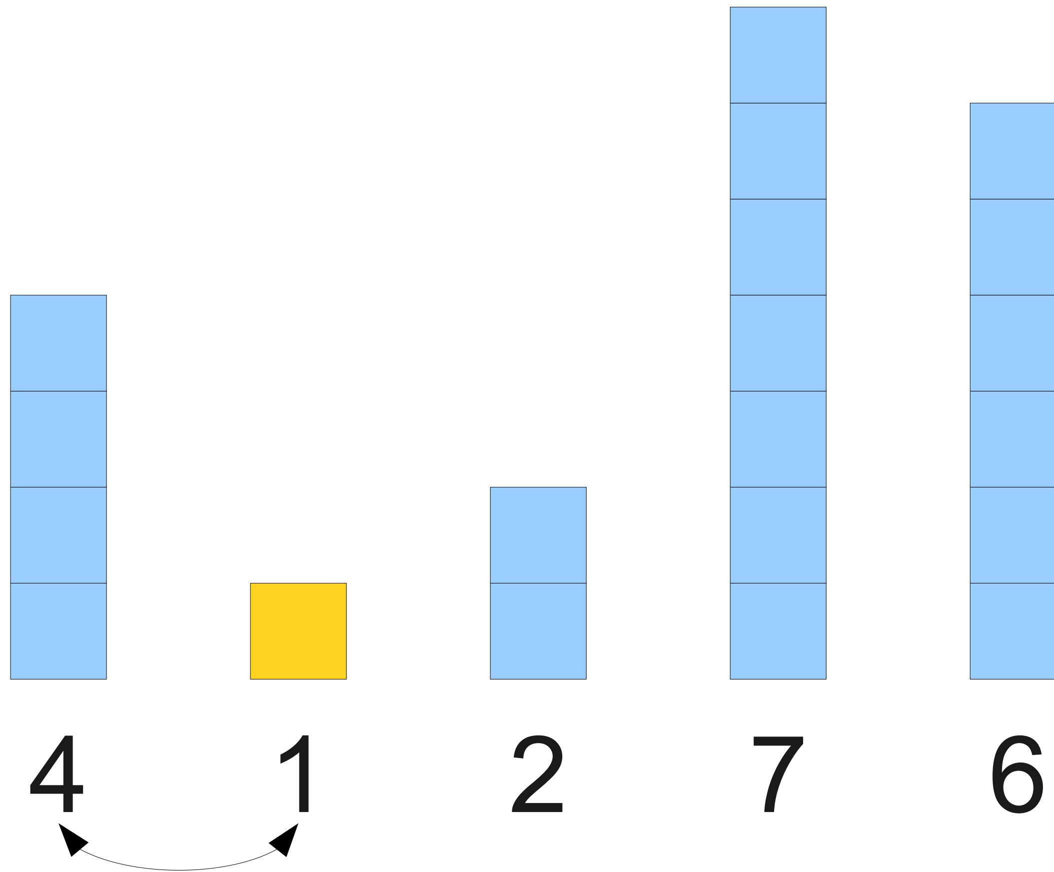
An Initial Idea: **Selection Sort**



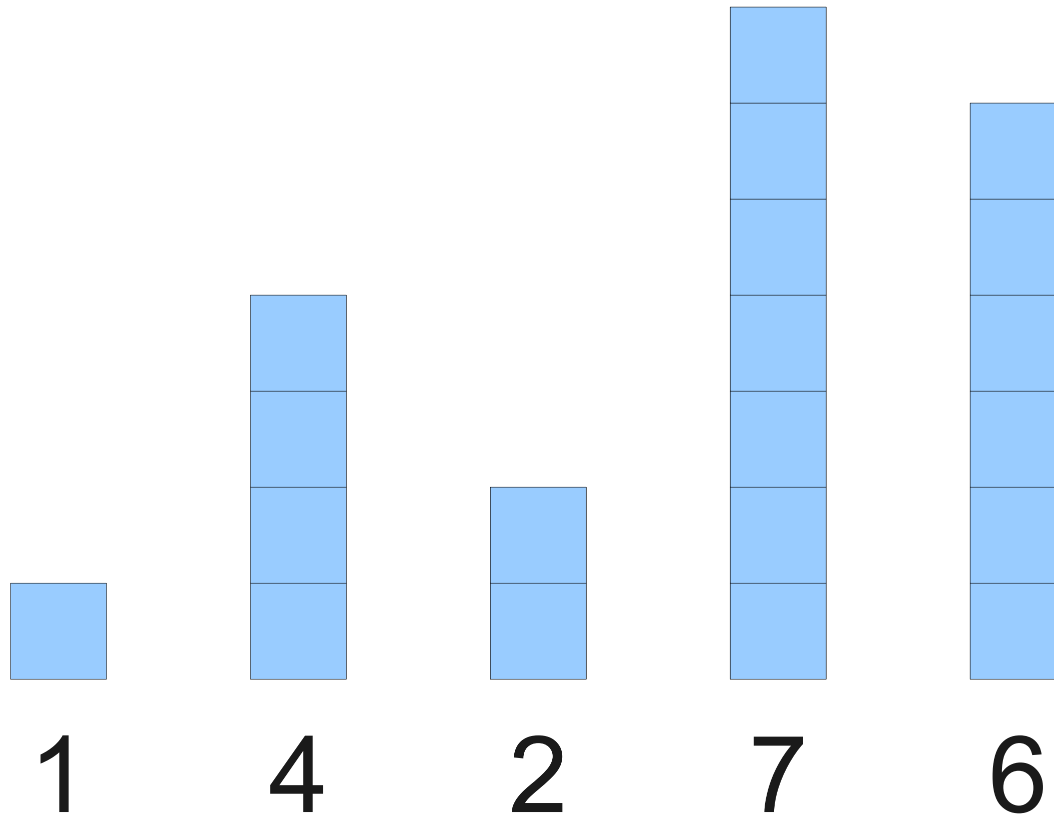
An Initial Idea: **Selection Sort**



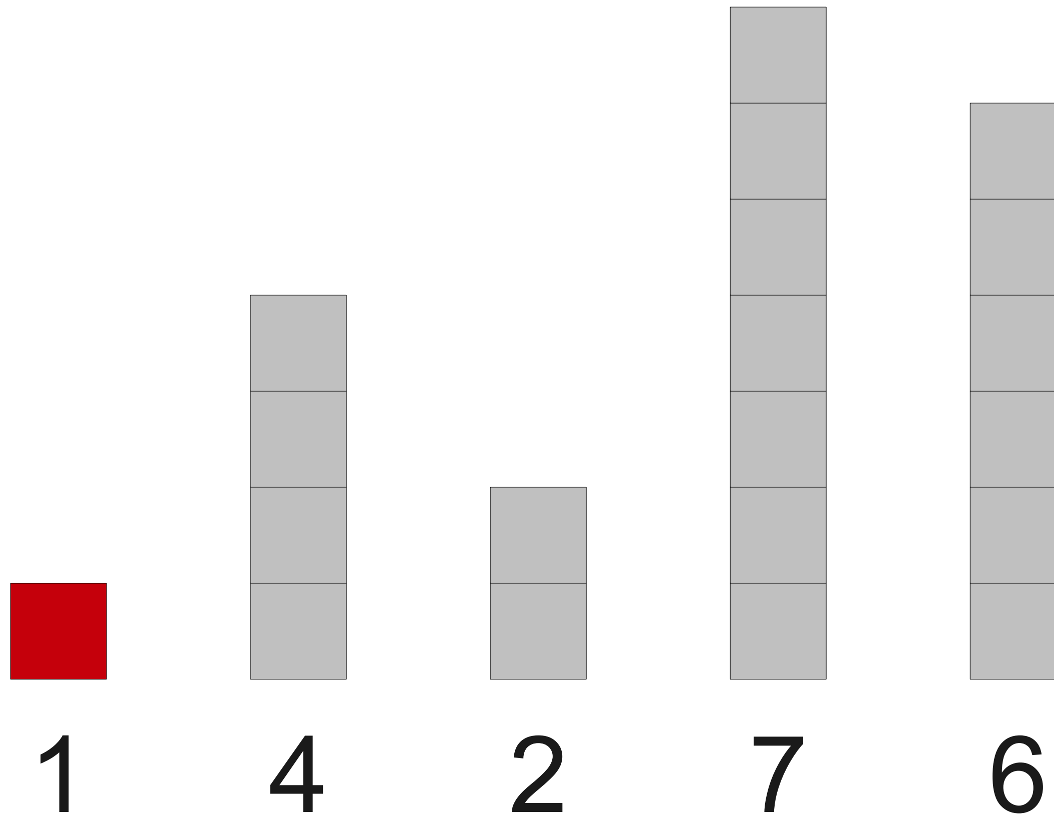
An Initial Idea: **Selection Sort**



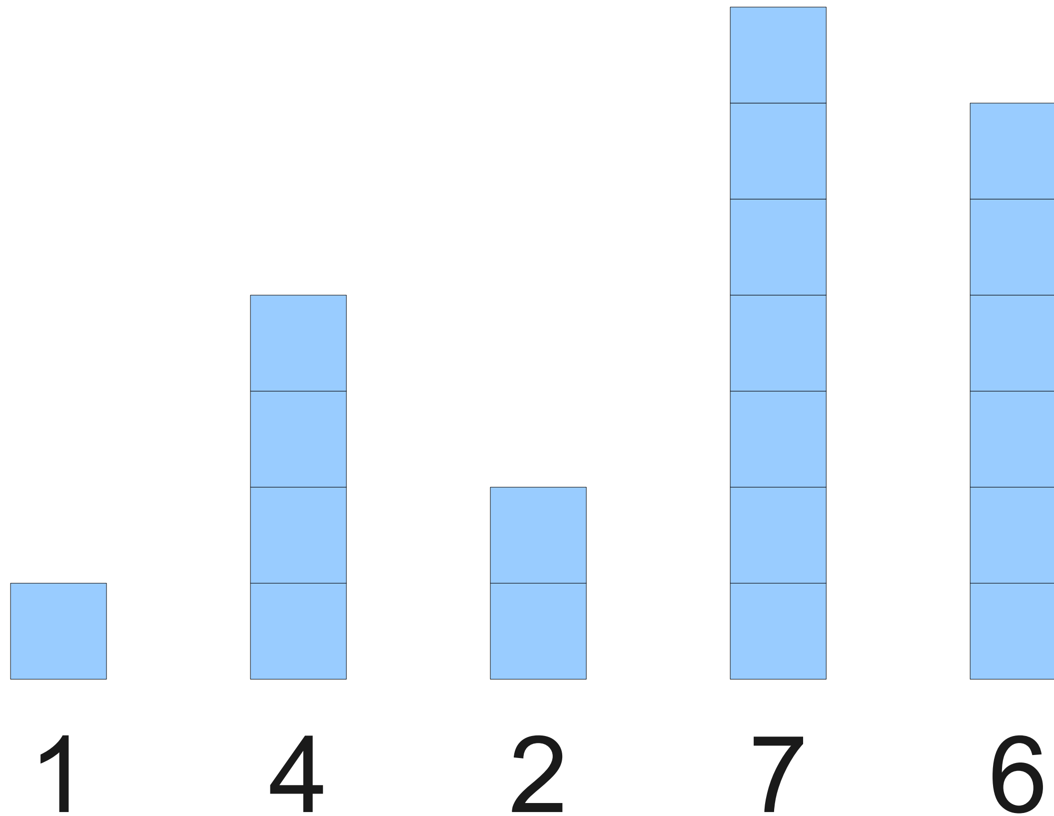
An Initial Idea: **Selection Sort**



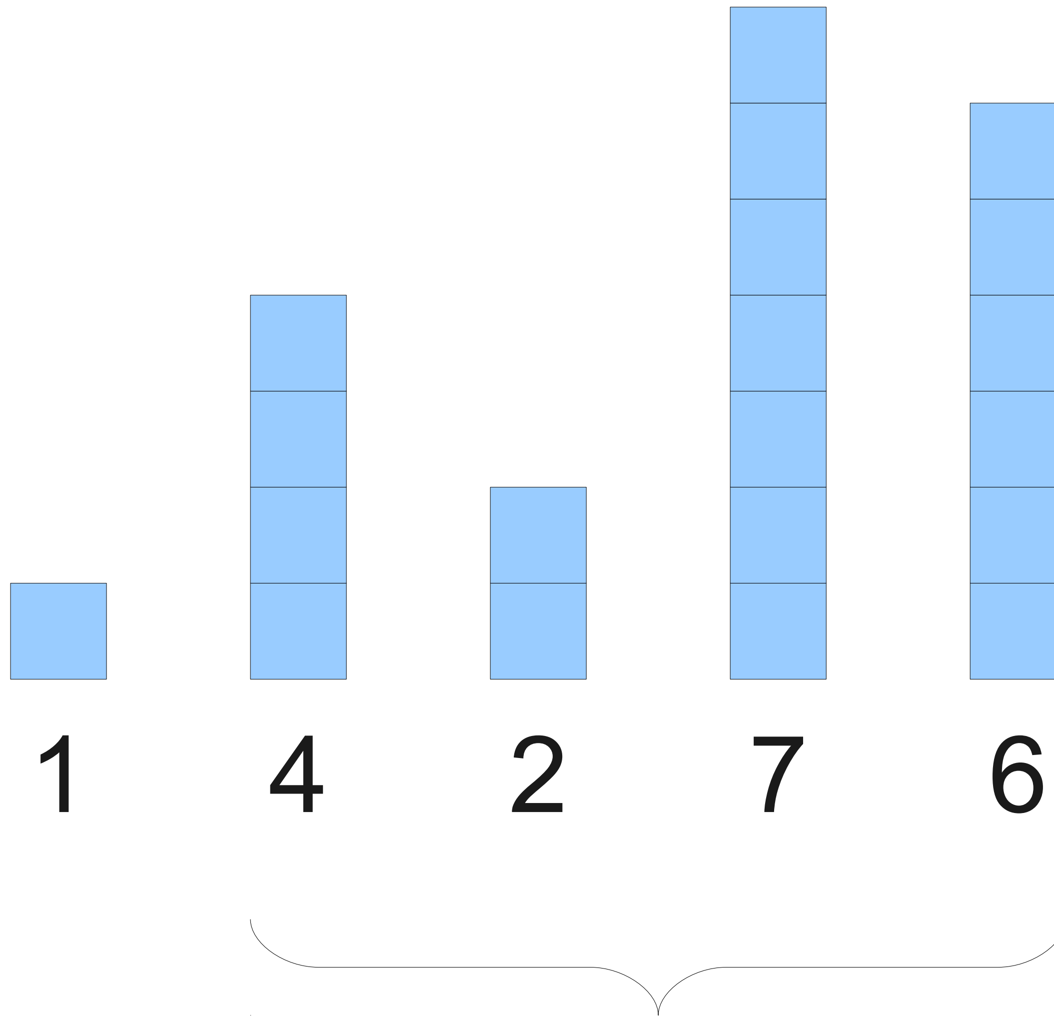
An Initial Idea: **Selection Sort**



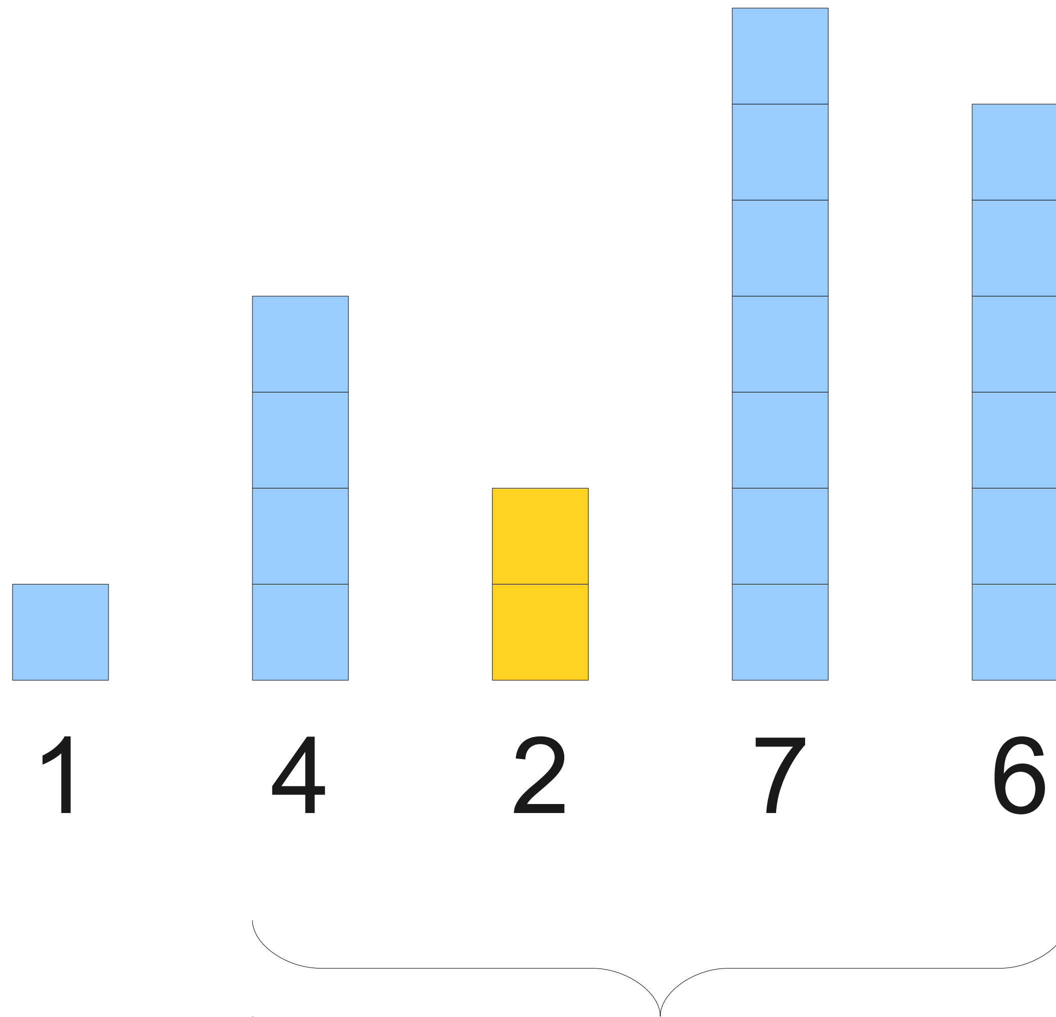
An Initial Idea: **Selection Sort**



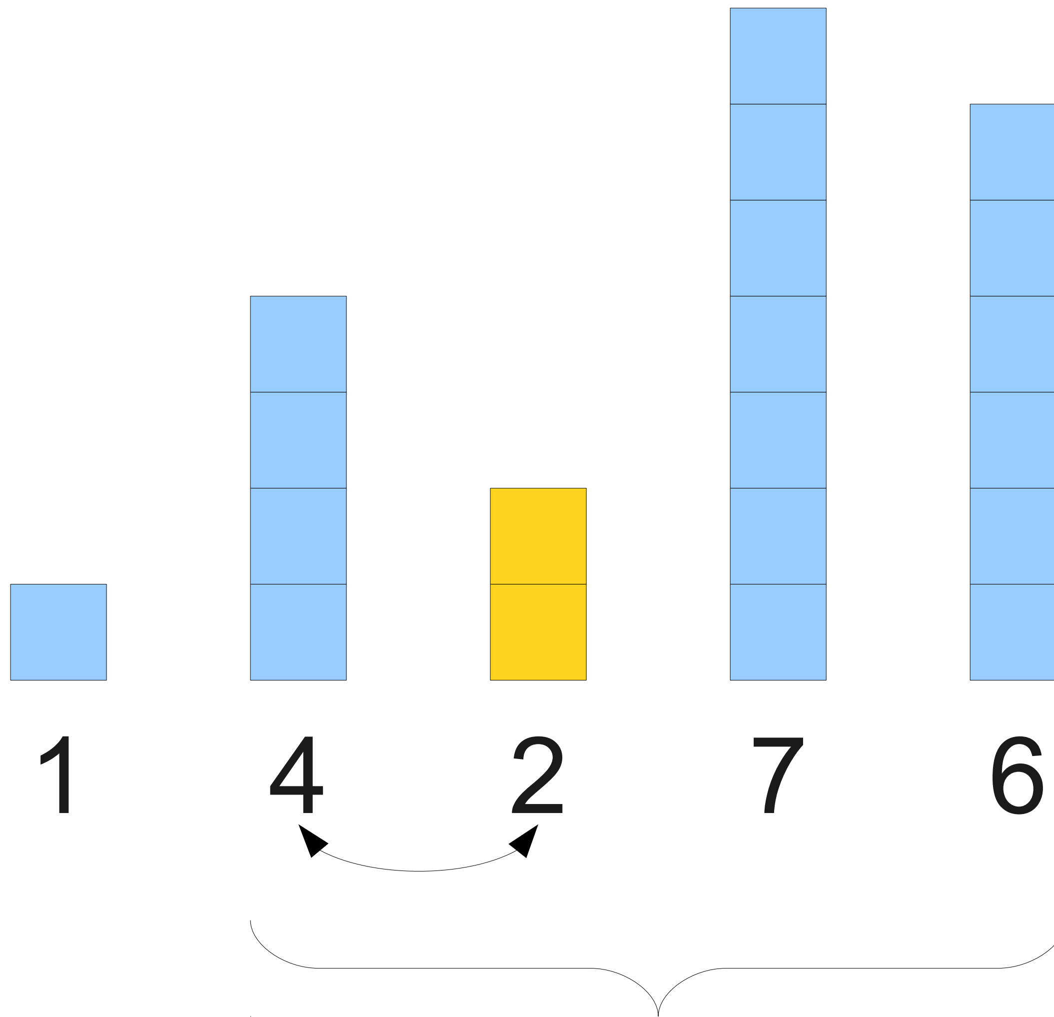
An Initial Idea: **Selection Sort**



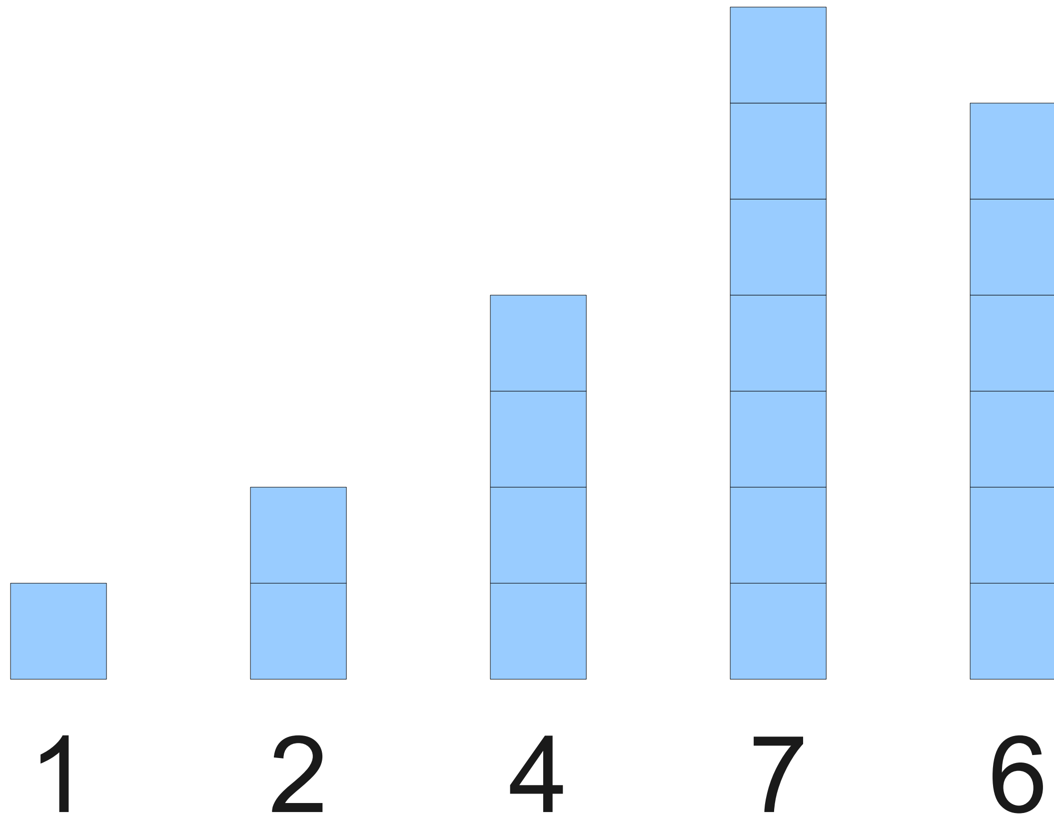
An Initial Idea: **Selection Sort**



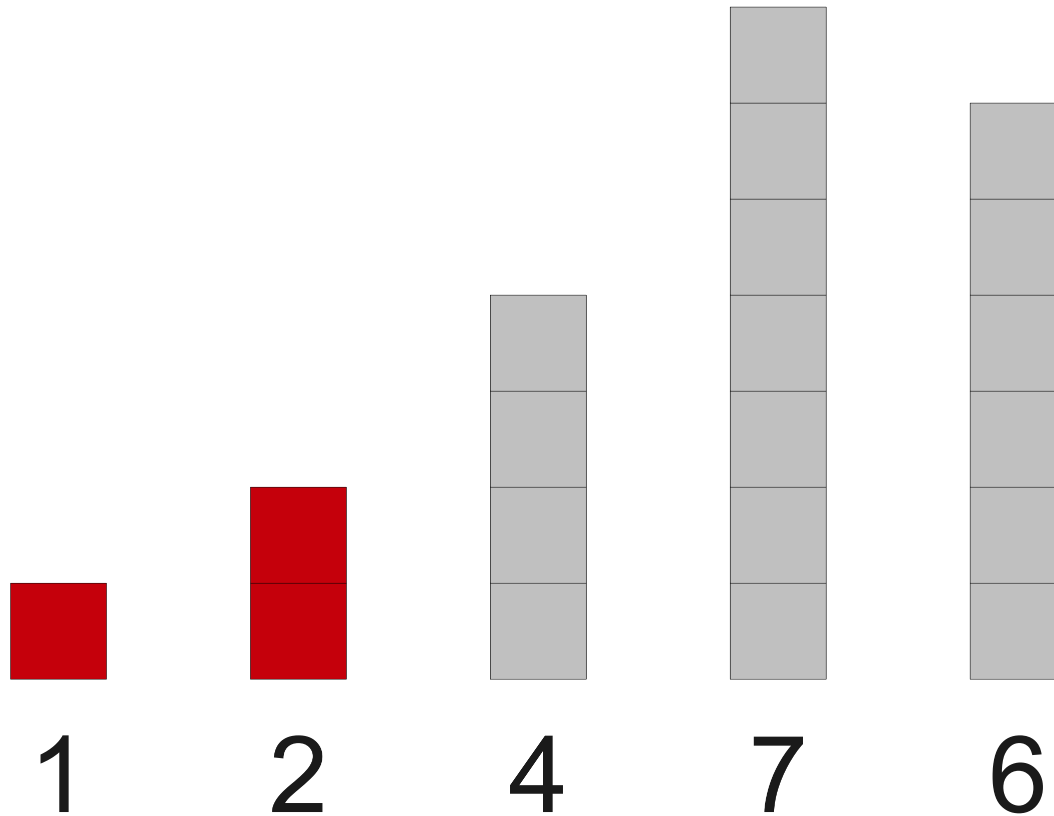
An Initial Idea: **Selection Sort**



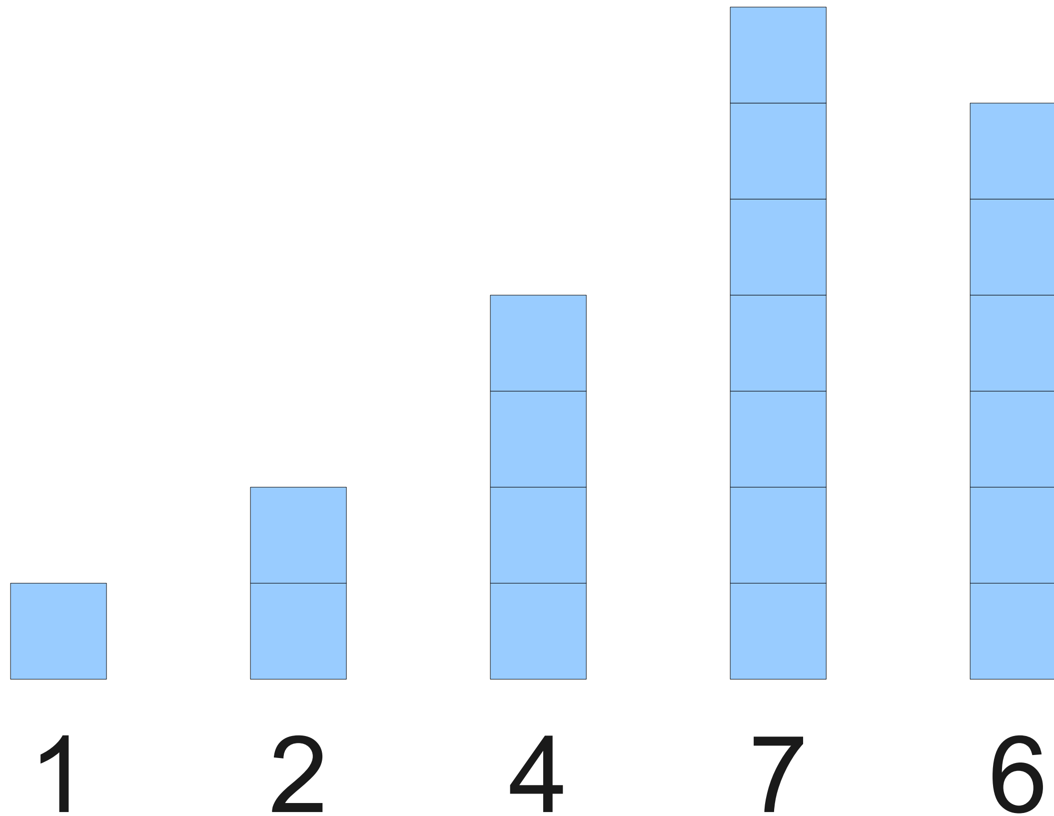
An Initial Idea: **Selection Sort**



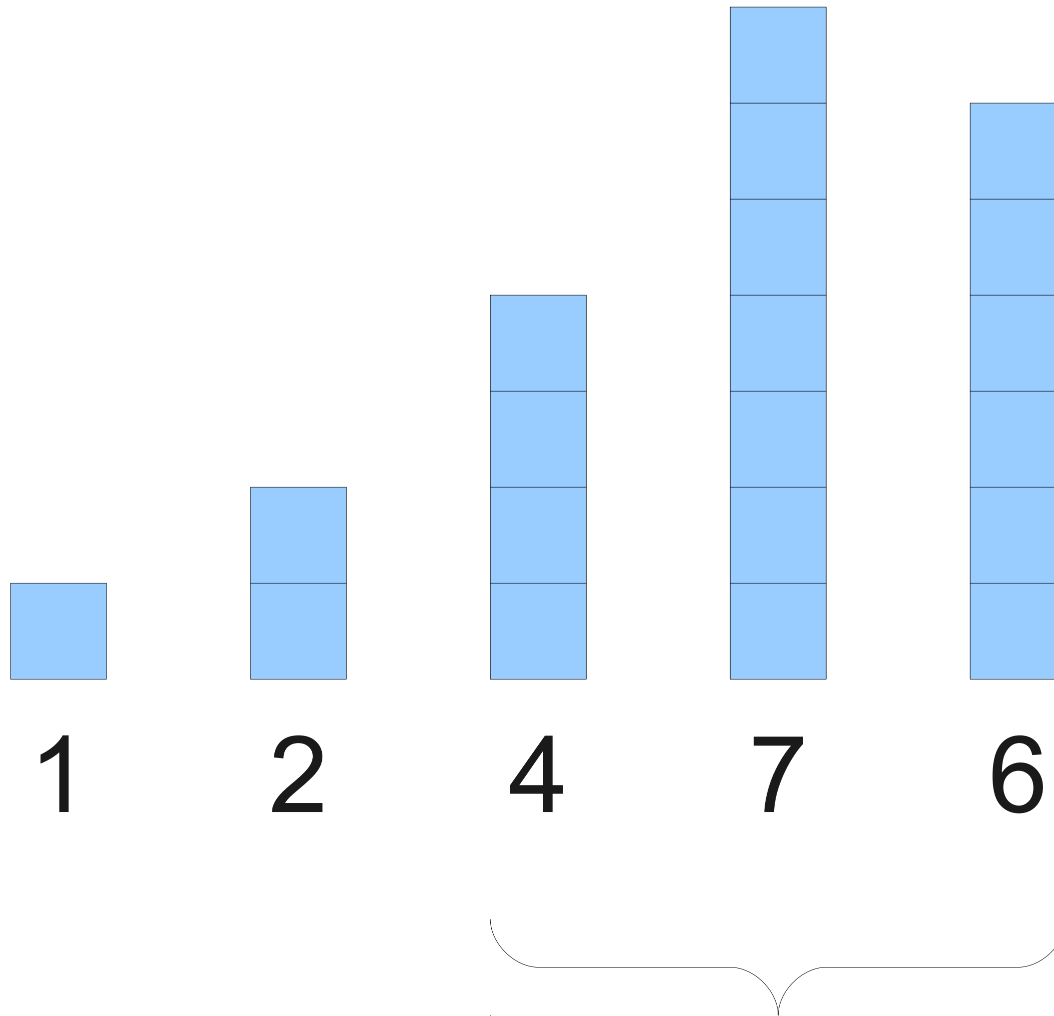
An Initial Idea: **Selection Sort**



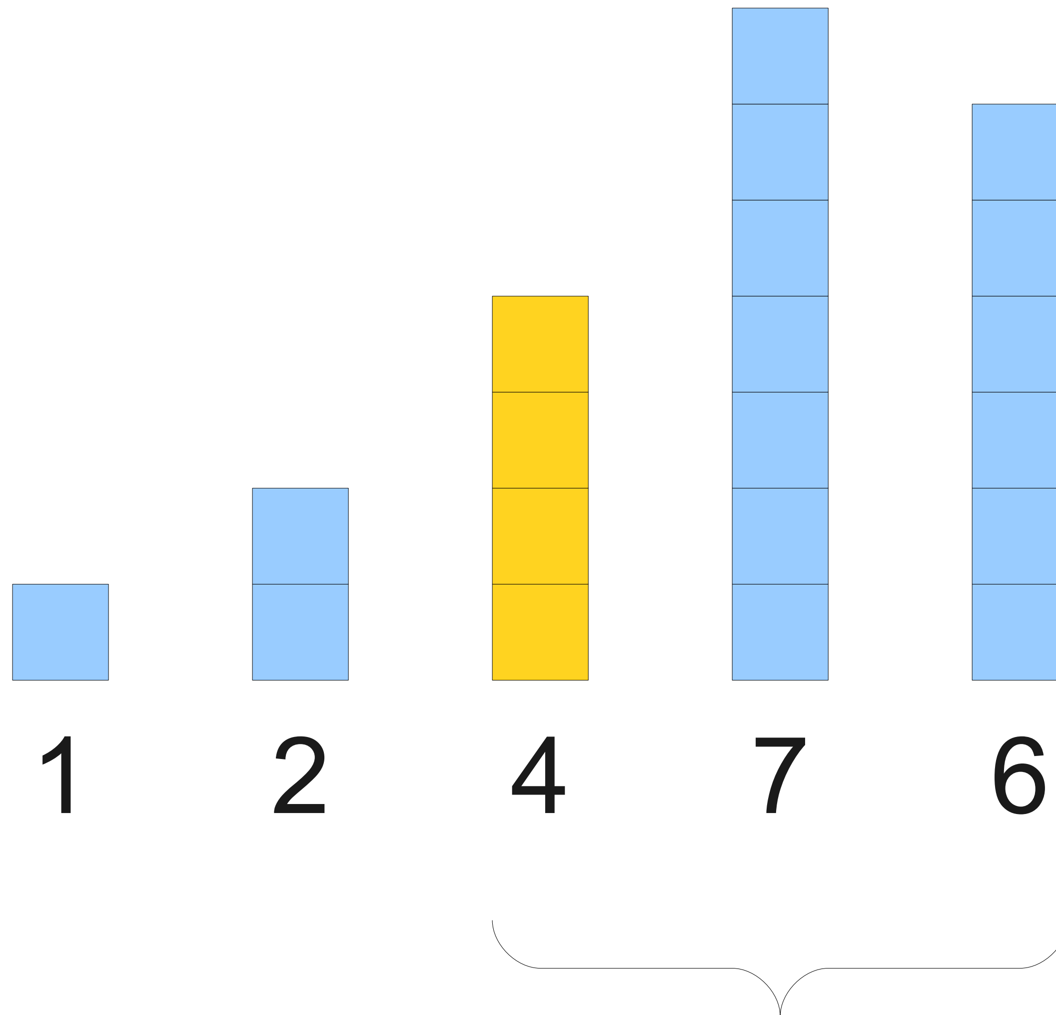
An Initial Idea: **Selection Sort**



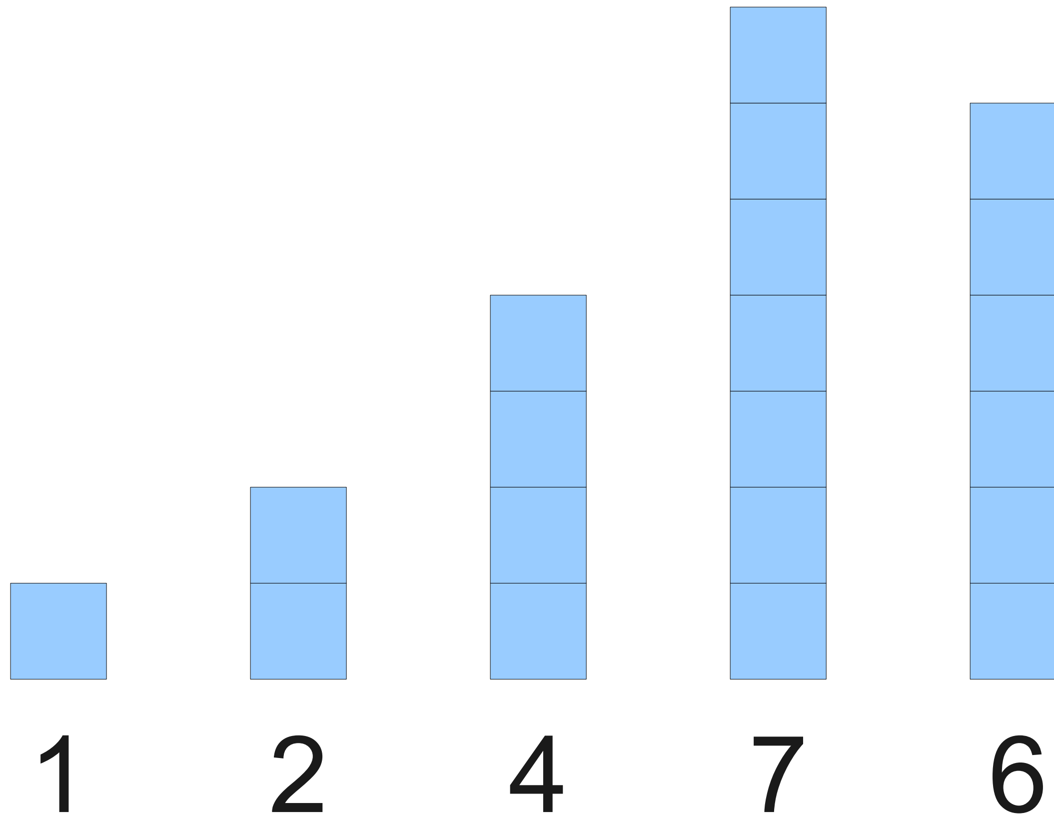
An Initial Idea: **Selection Sort**



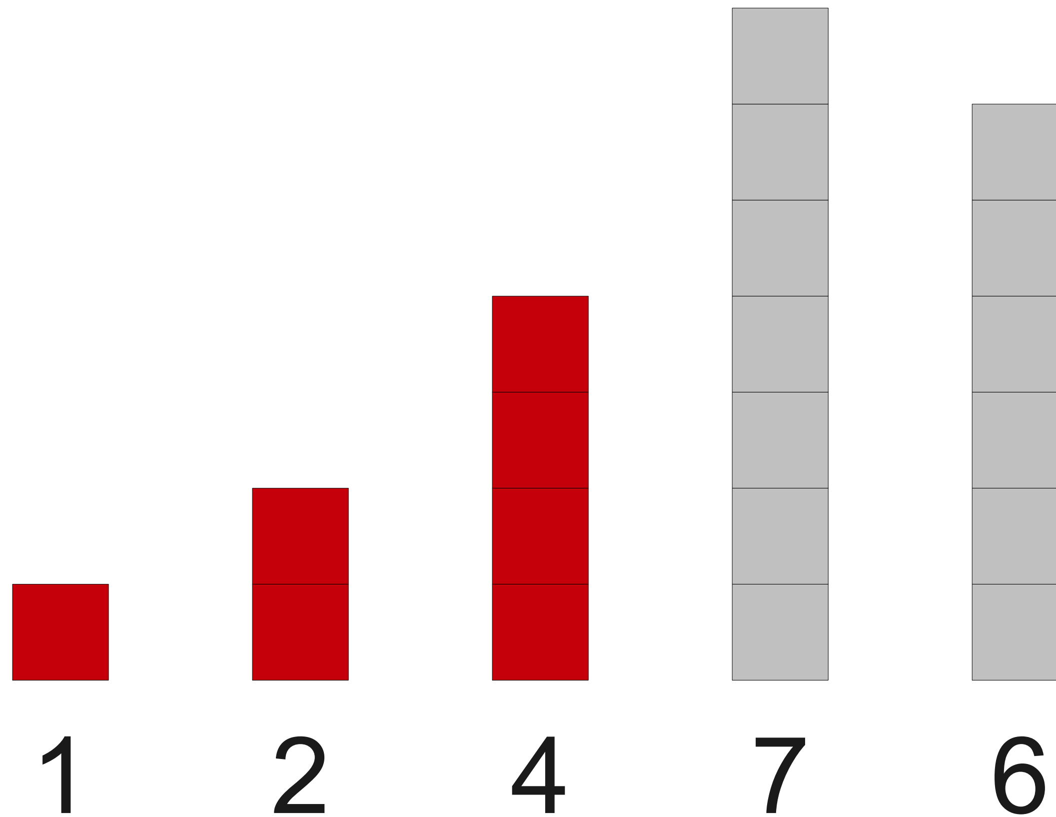
An Initial Idea: **Selection Sort**



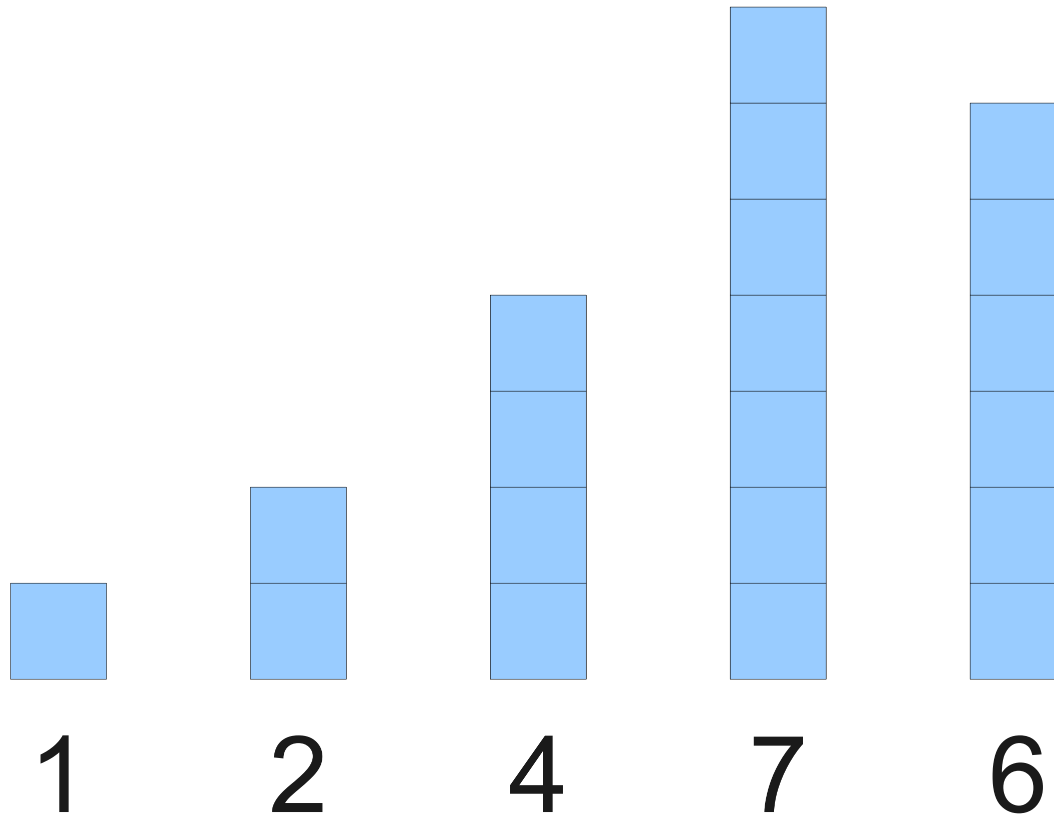
An Initial Idea: **Selection Sort**



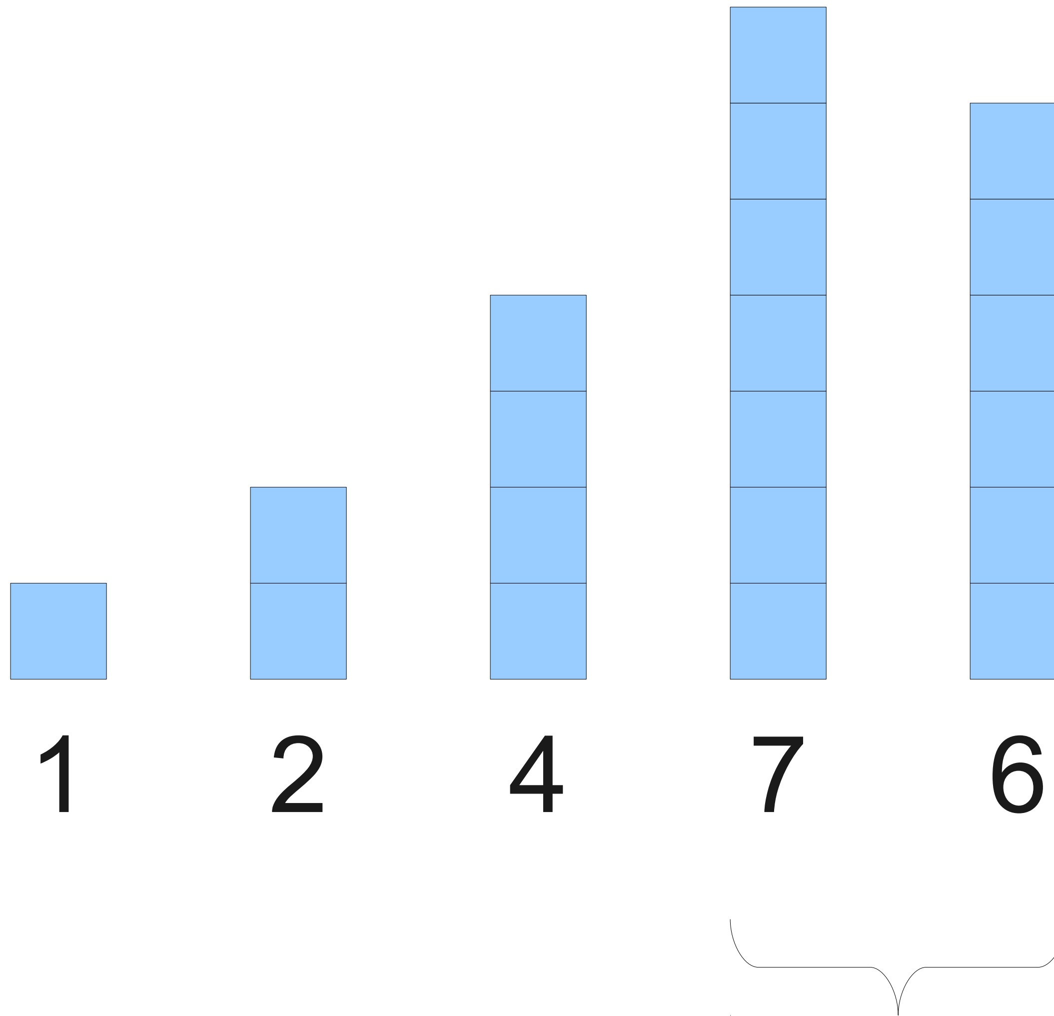
An Initial Idea: **Selection Sort**



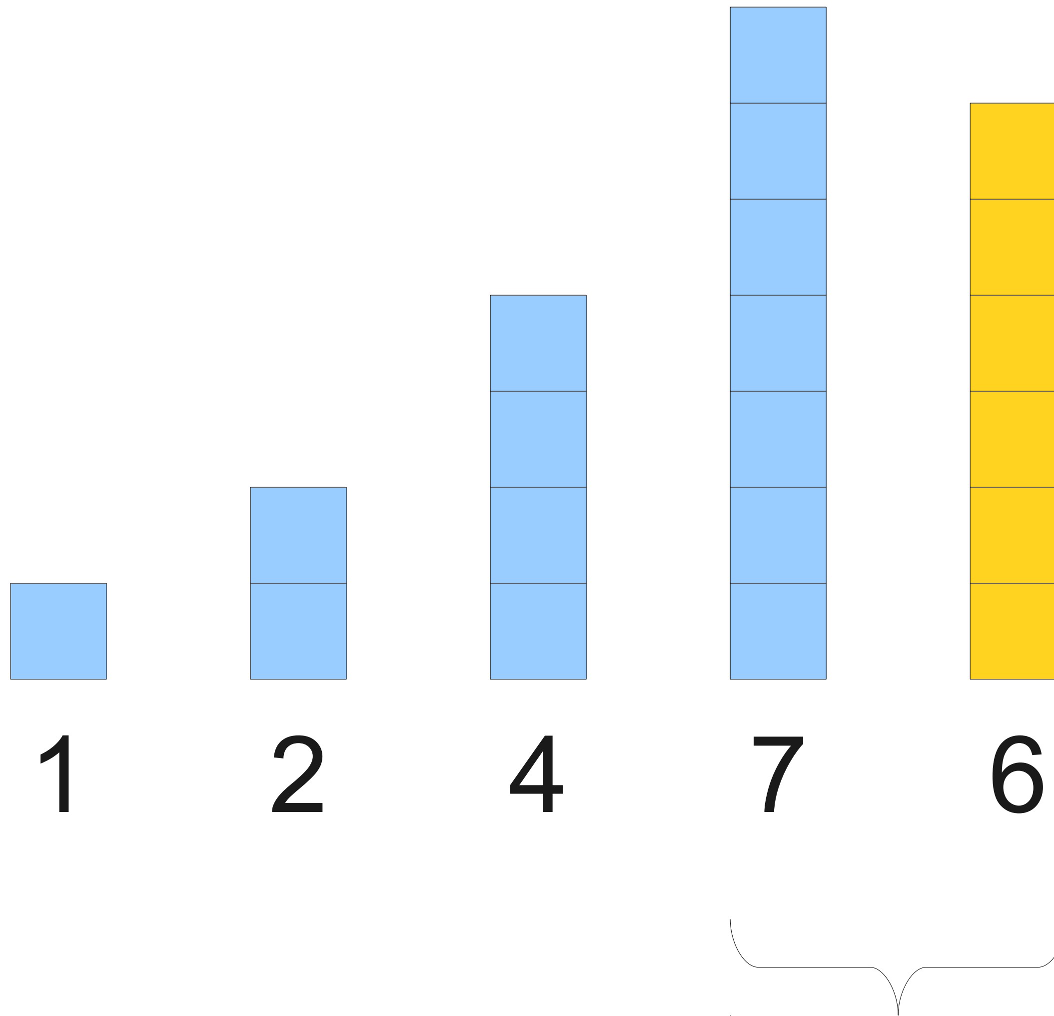
An Initial Idea: **Selection Sort**



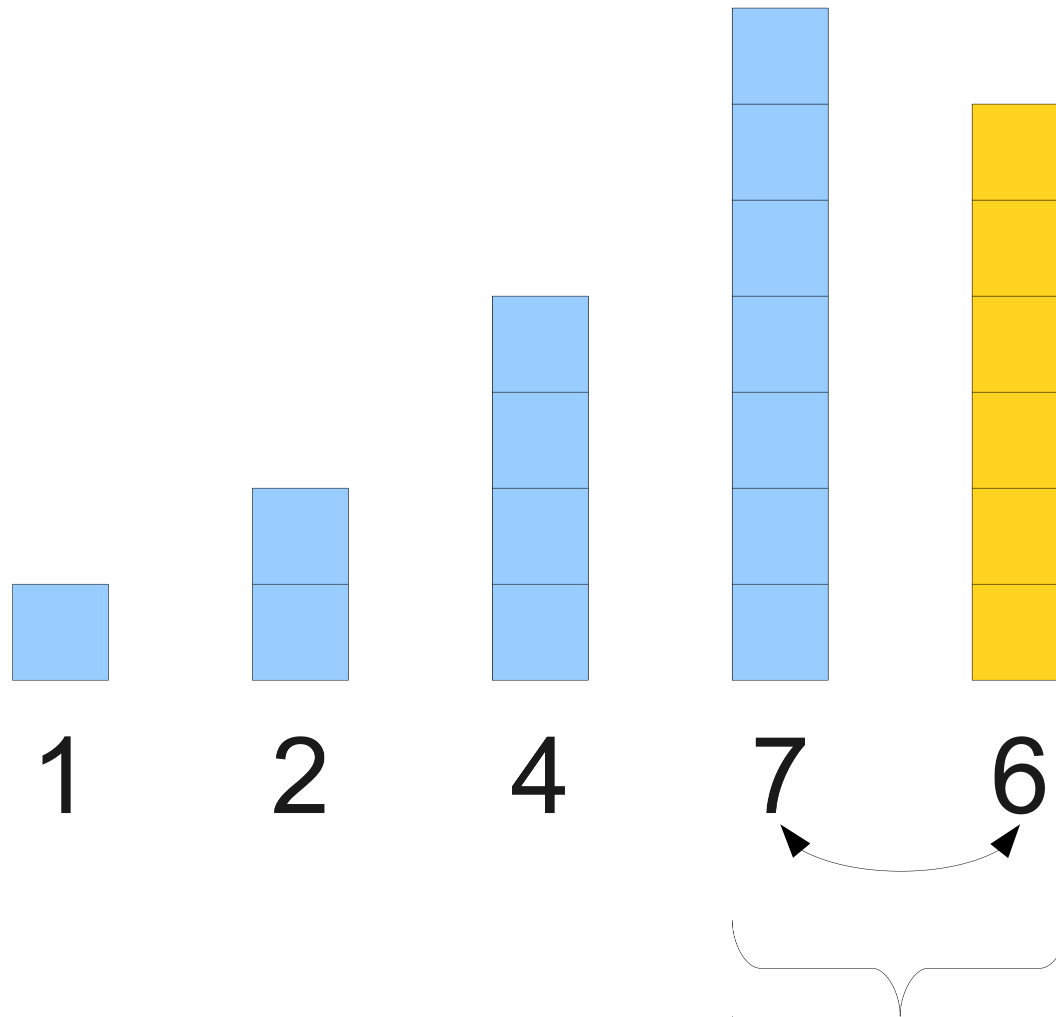
An Initial Idea: **Selection Sort**



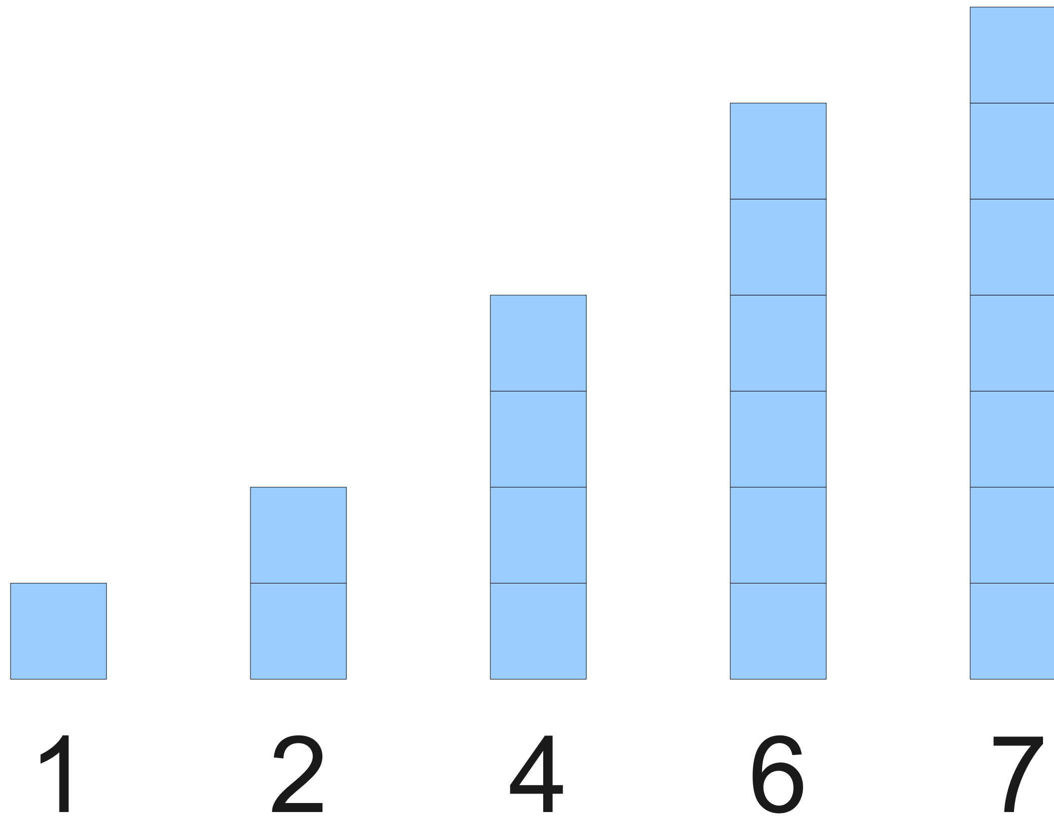
An Initial Idea: **Selection Sort**



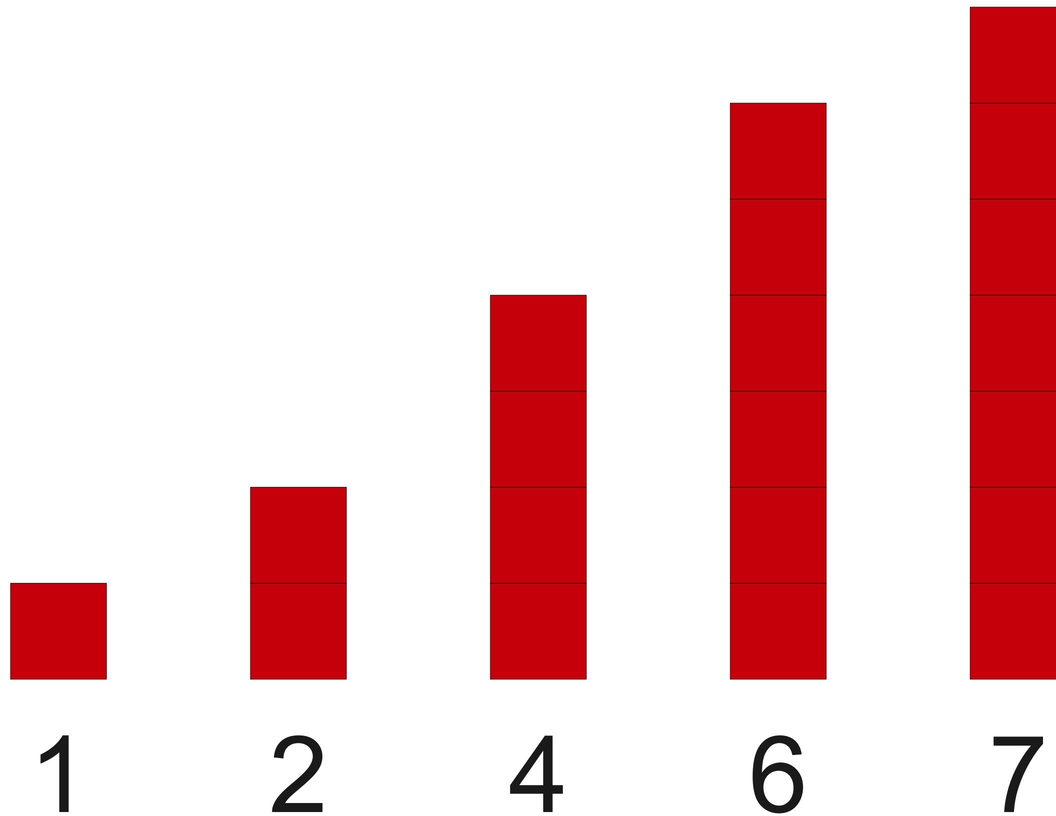
An Initial Idea: **Selection Sort**



An Initial Idea: **Selection Sort**



An Initial Idea: **Selection Sort**

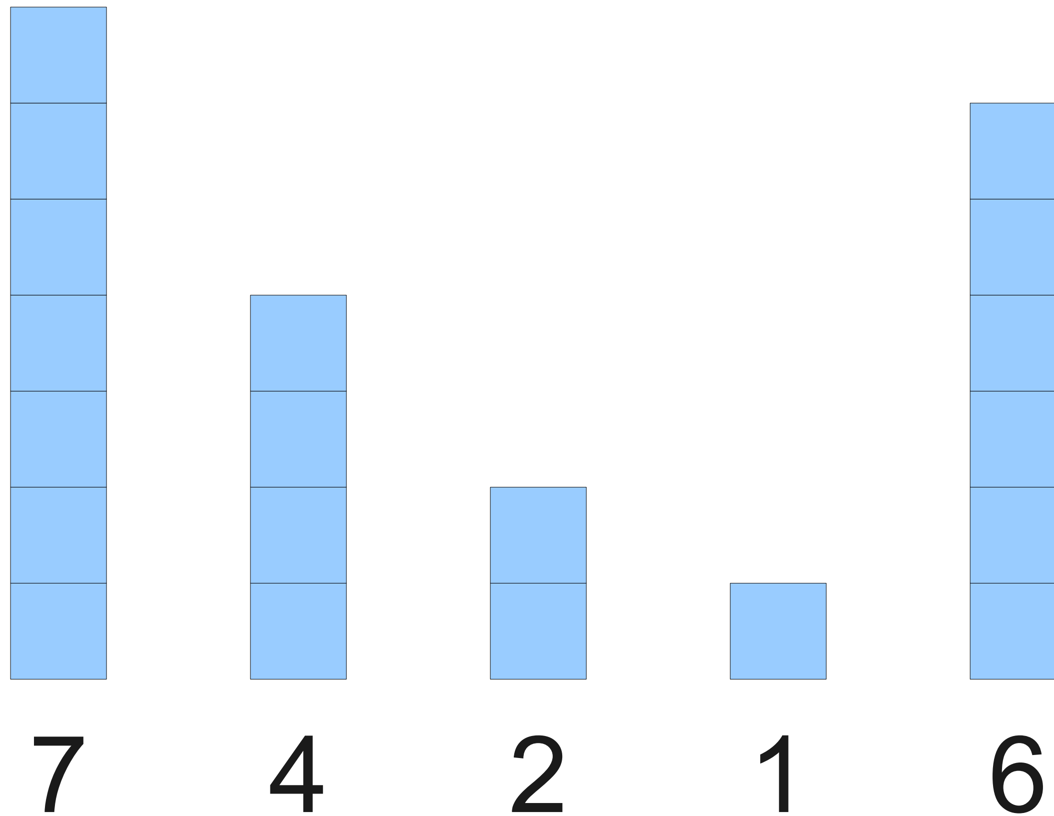


Notes on Selection Sort

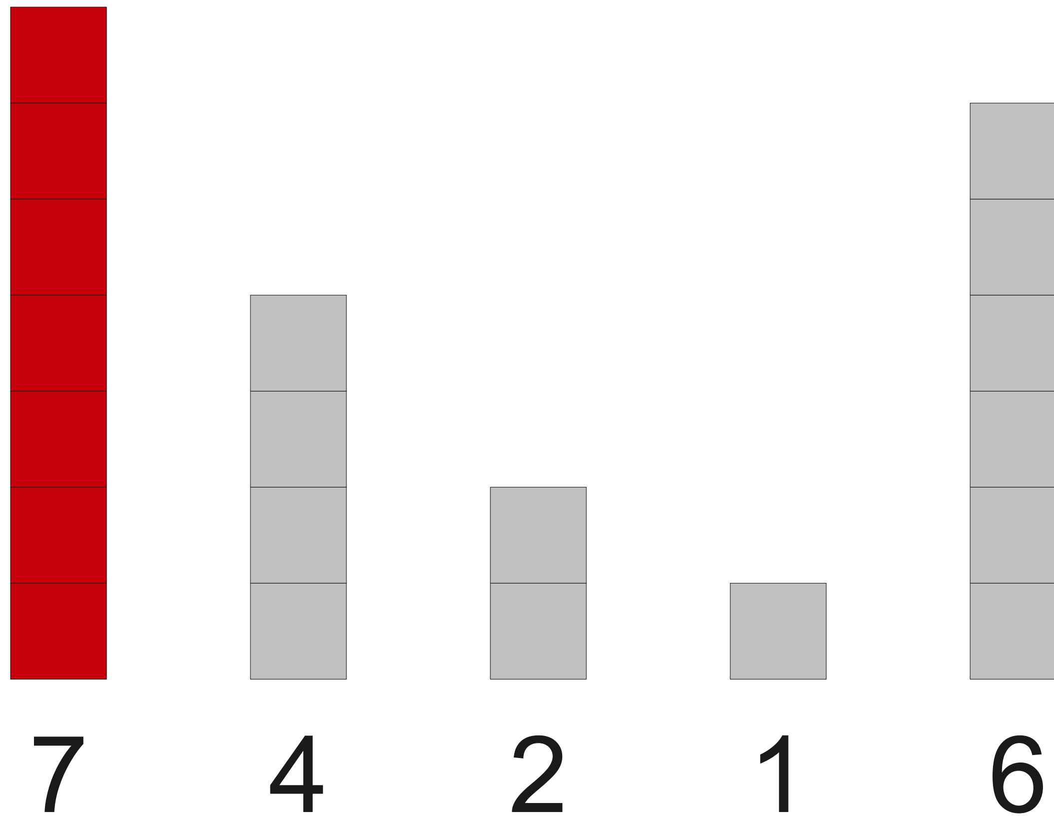
- Selection sort is $O(n^2)$ in the worst case.
- Selection sort is $O(n^2)$ in the best case.
- Notation: Selection sort is $\Theta(n^2)$.

Another Idea: **Insertion Sort**

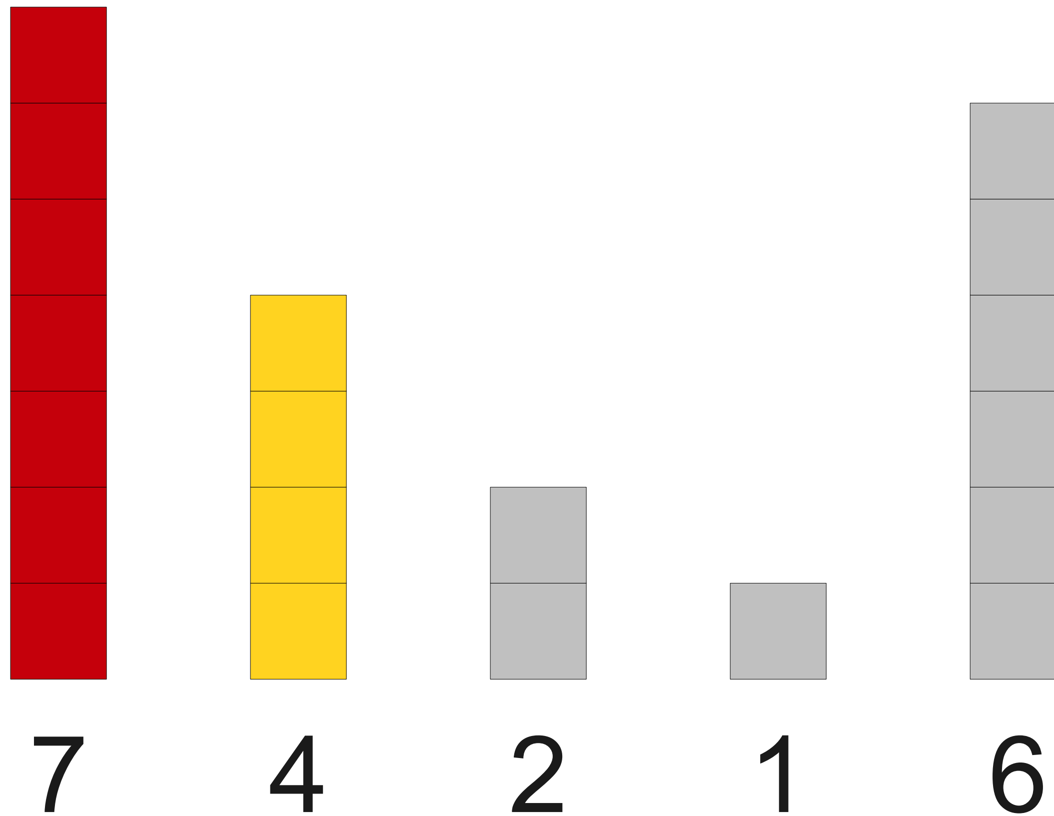
Another Idea: **Insertion Sort**



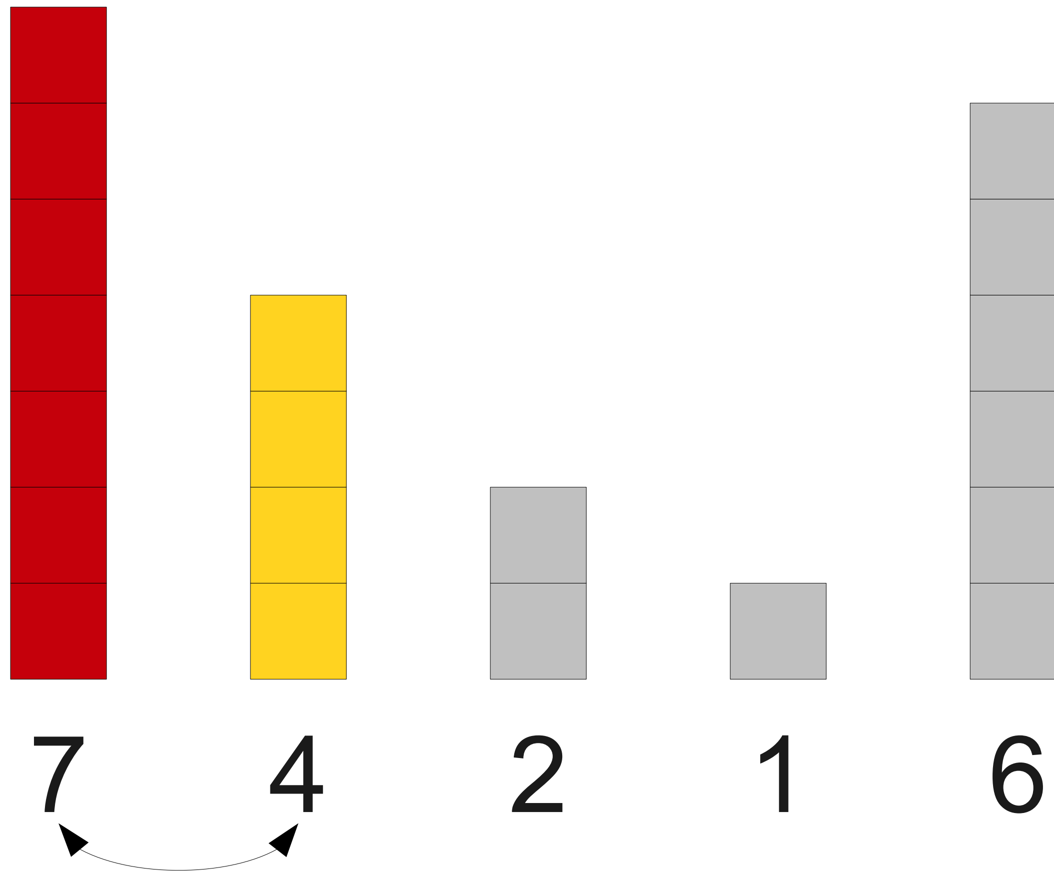
Another Idea: **Insertion Sort**



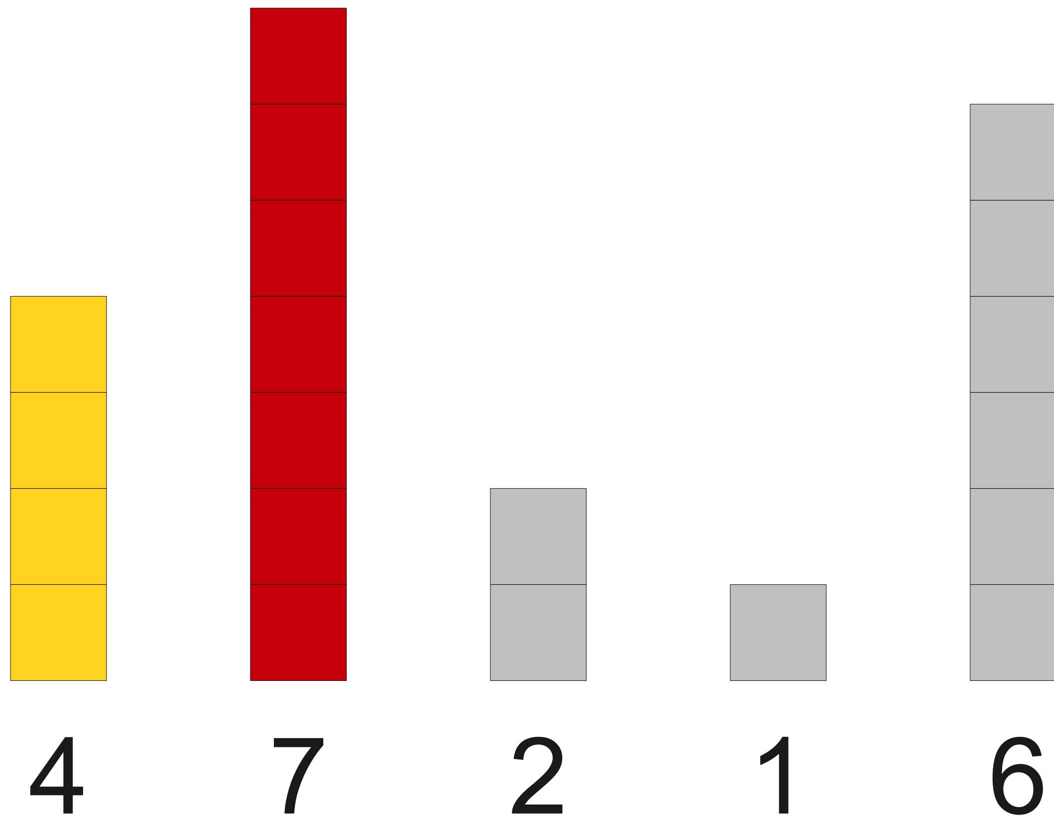
Another Idea: **Insertion Sort**



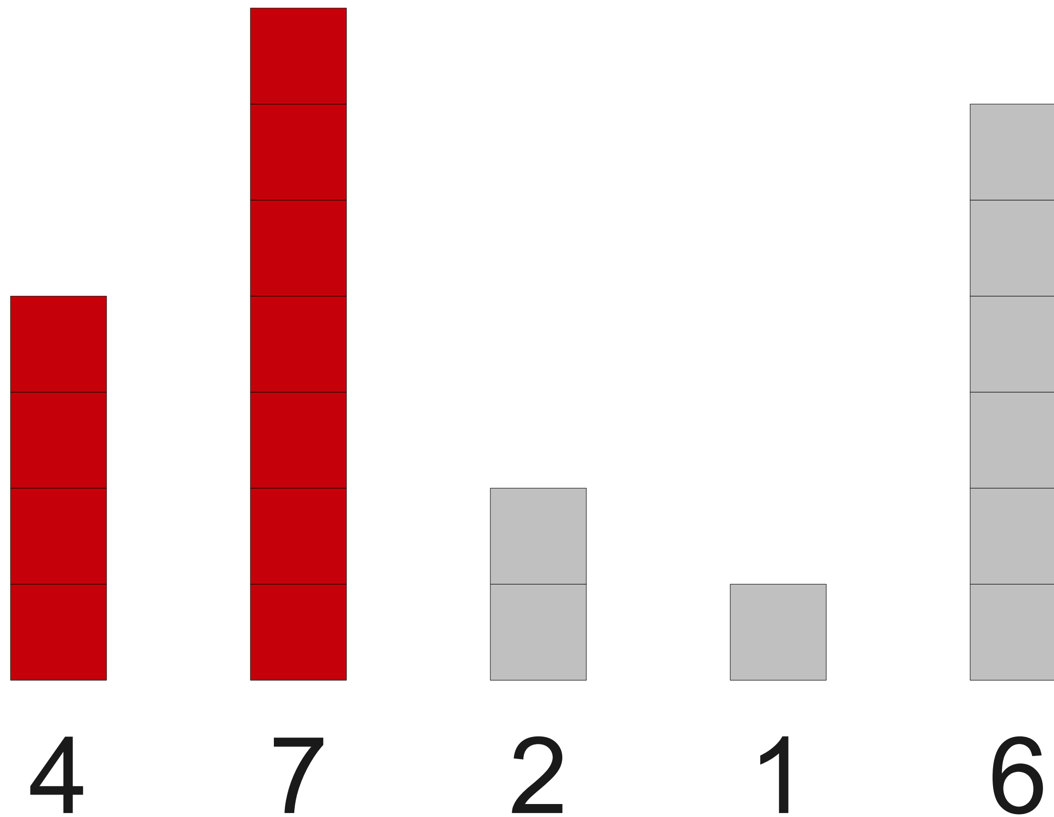
Another Idea: **Insertion Sort**



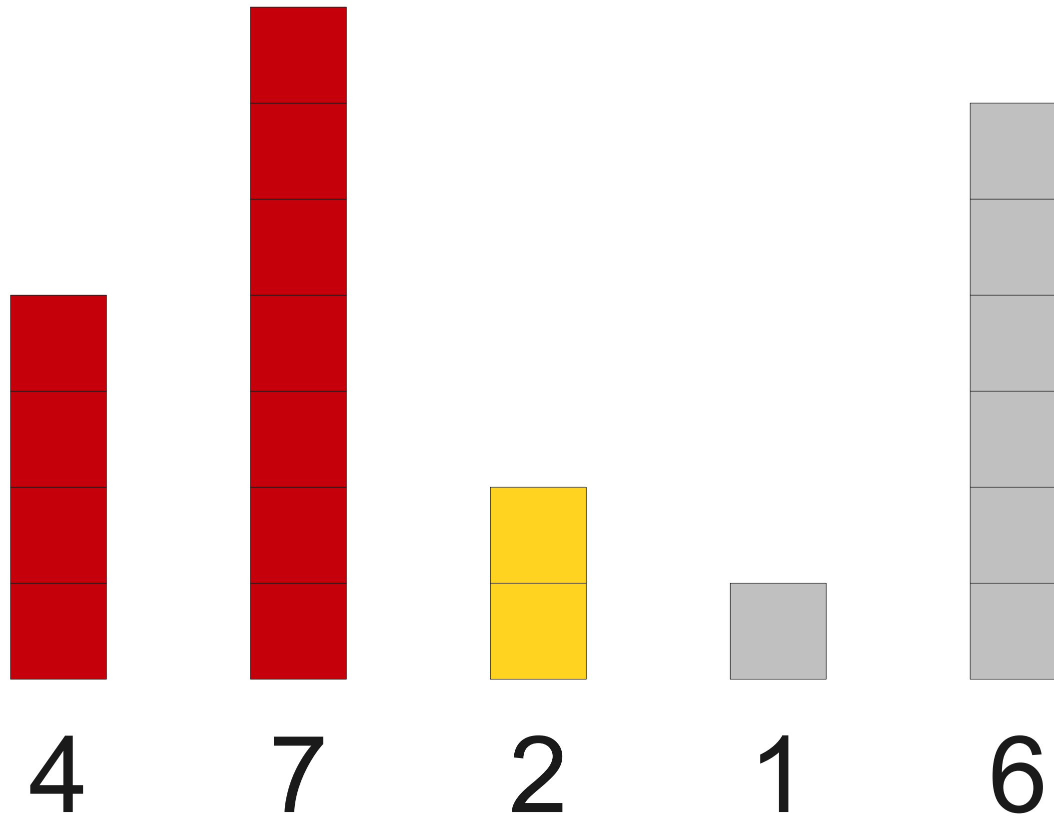
Another Idea: **Insertion Sort**



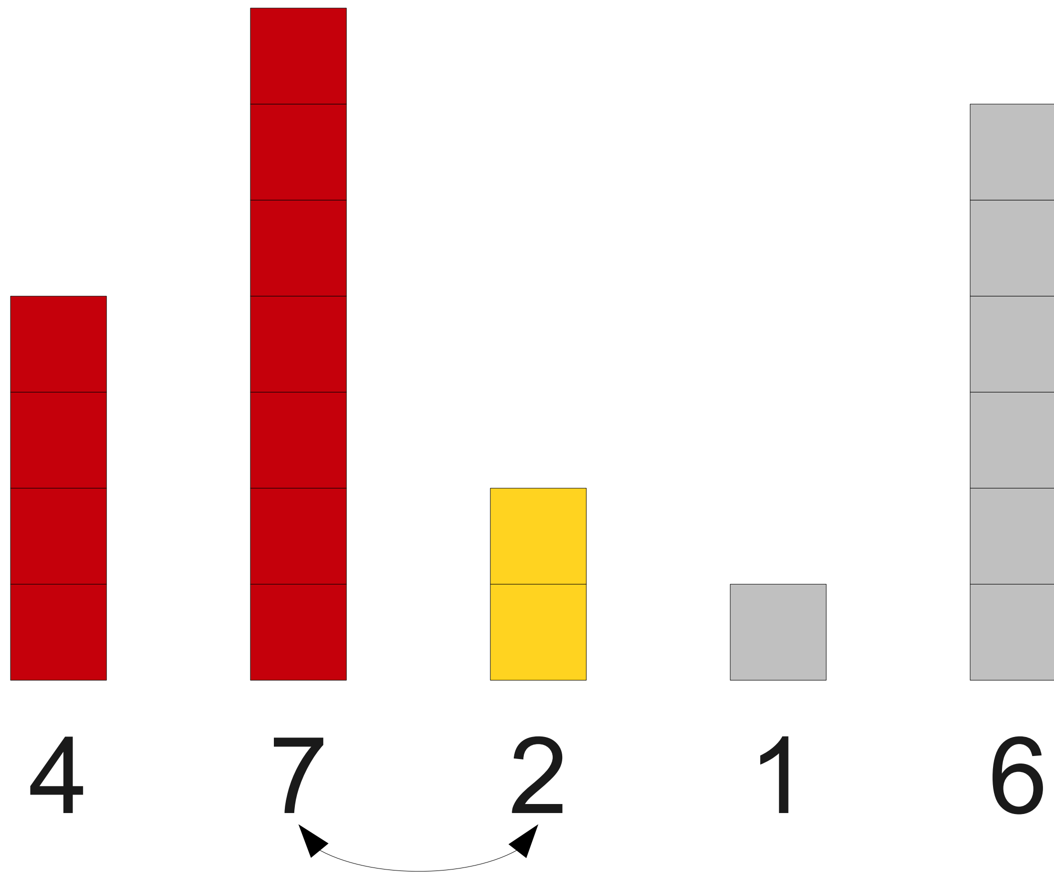
Another Idea: **Insertion Sort**



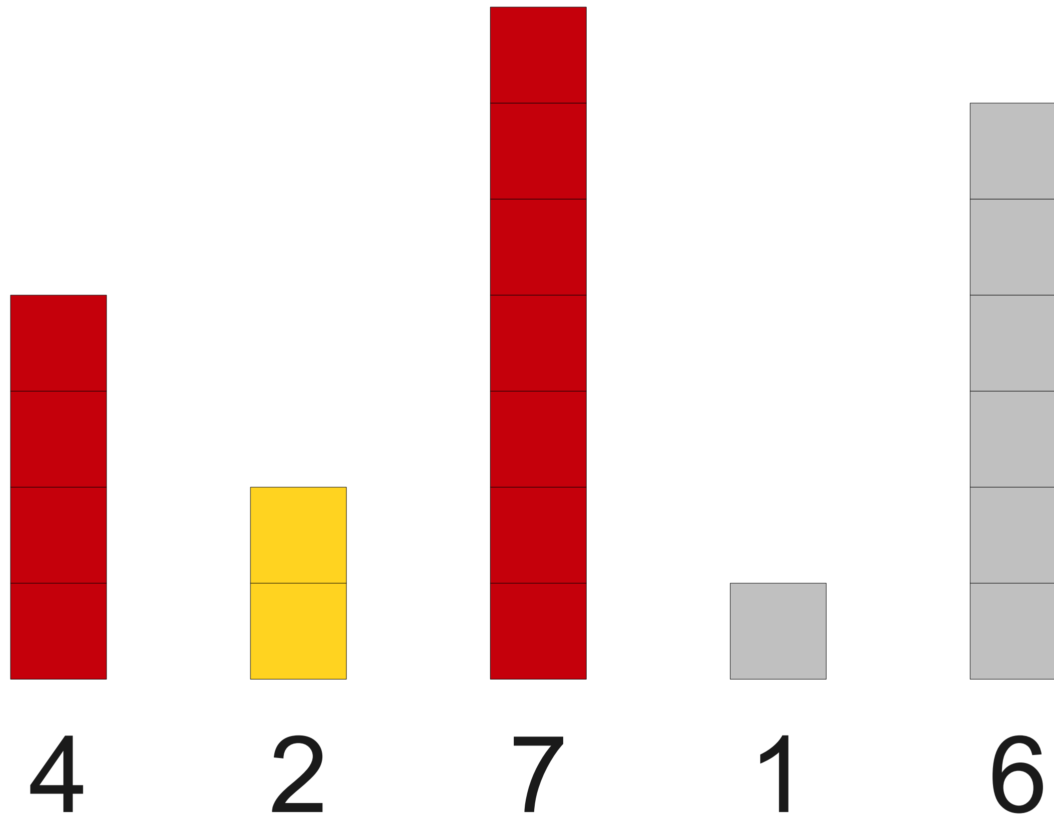
Another Idea: **Insertion Sort**



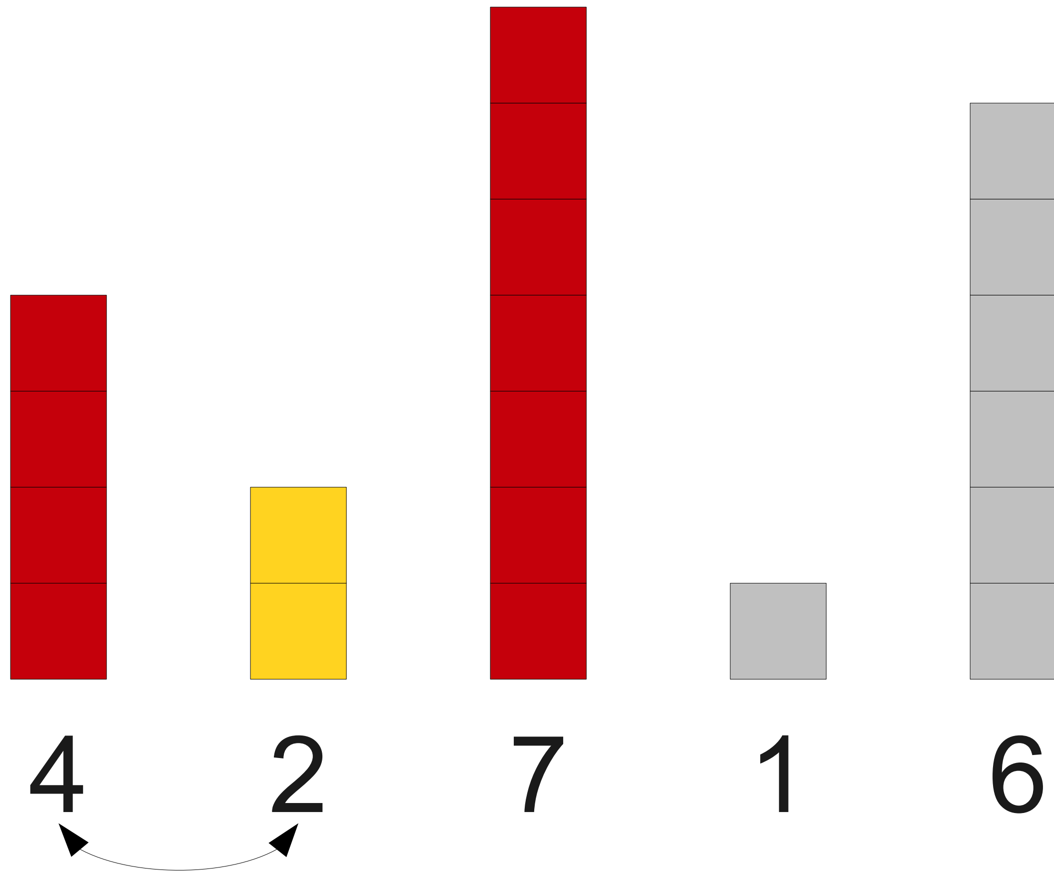
Another Idea: **Insertion Sort**



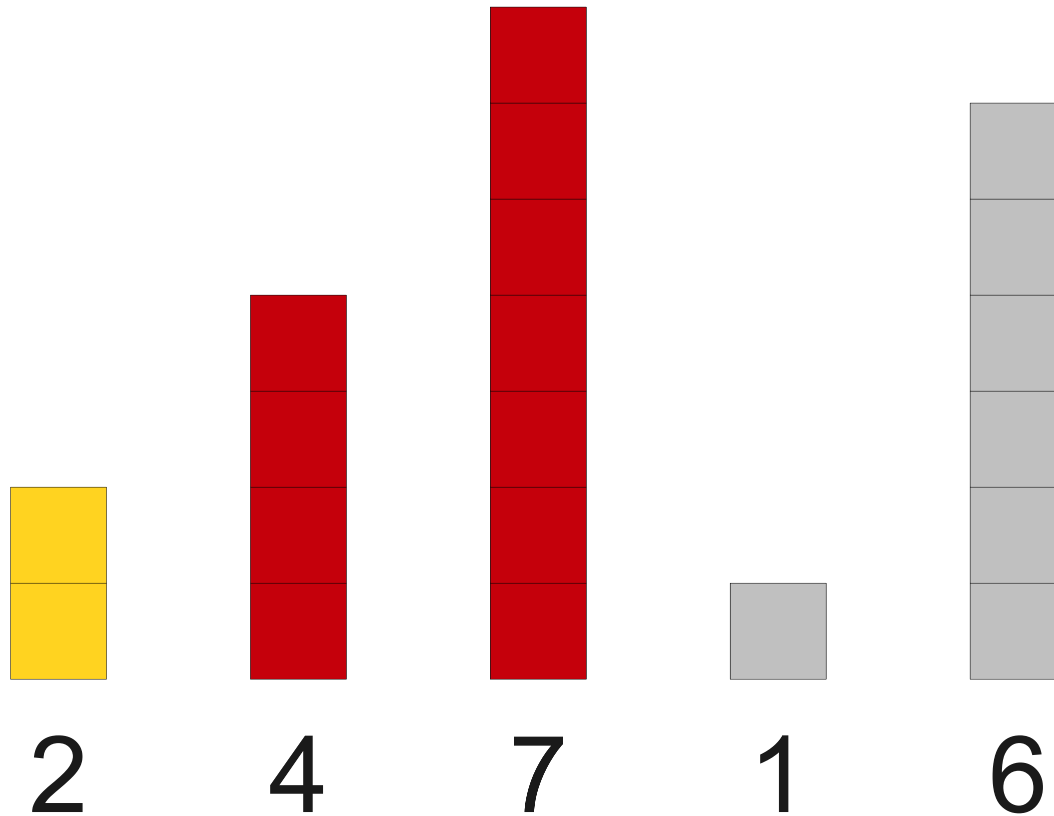
Another Idea: **Insertion Sort**



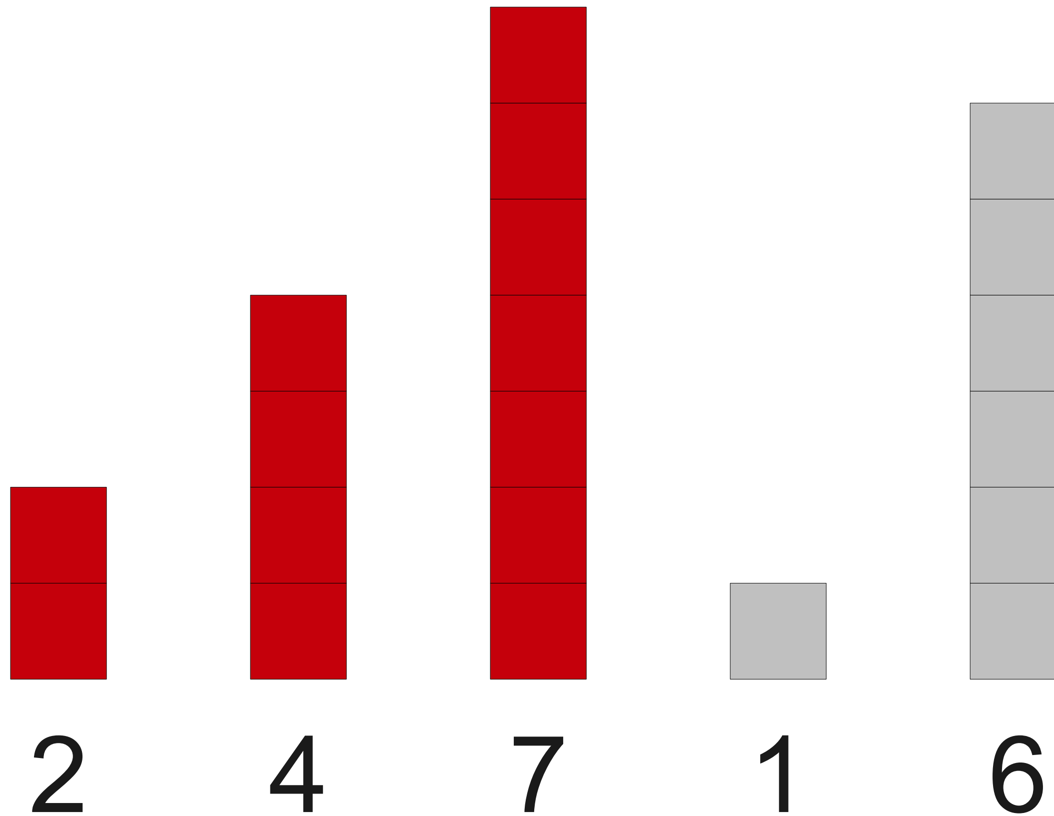
Another Idea: **Insertion Sort**



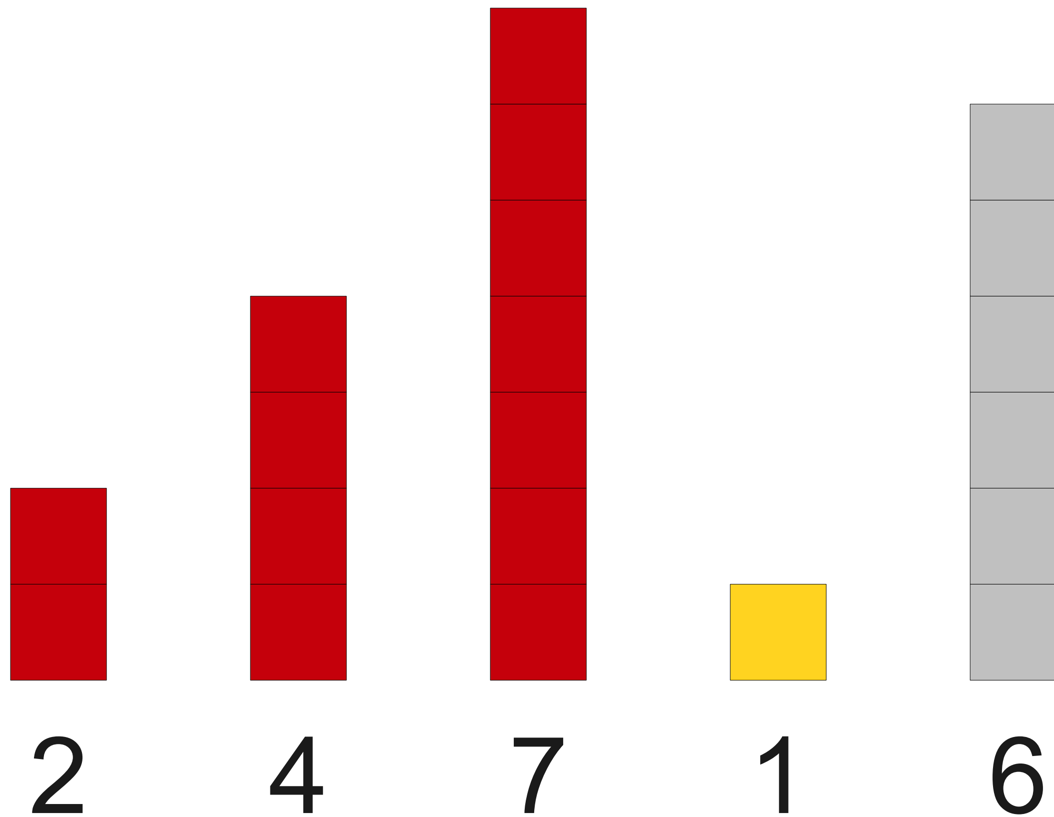
Another Idea: **Insertion Sort**



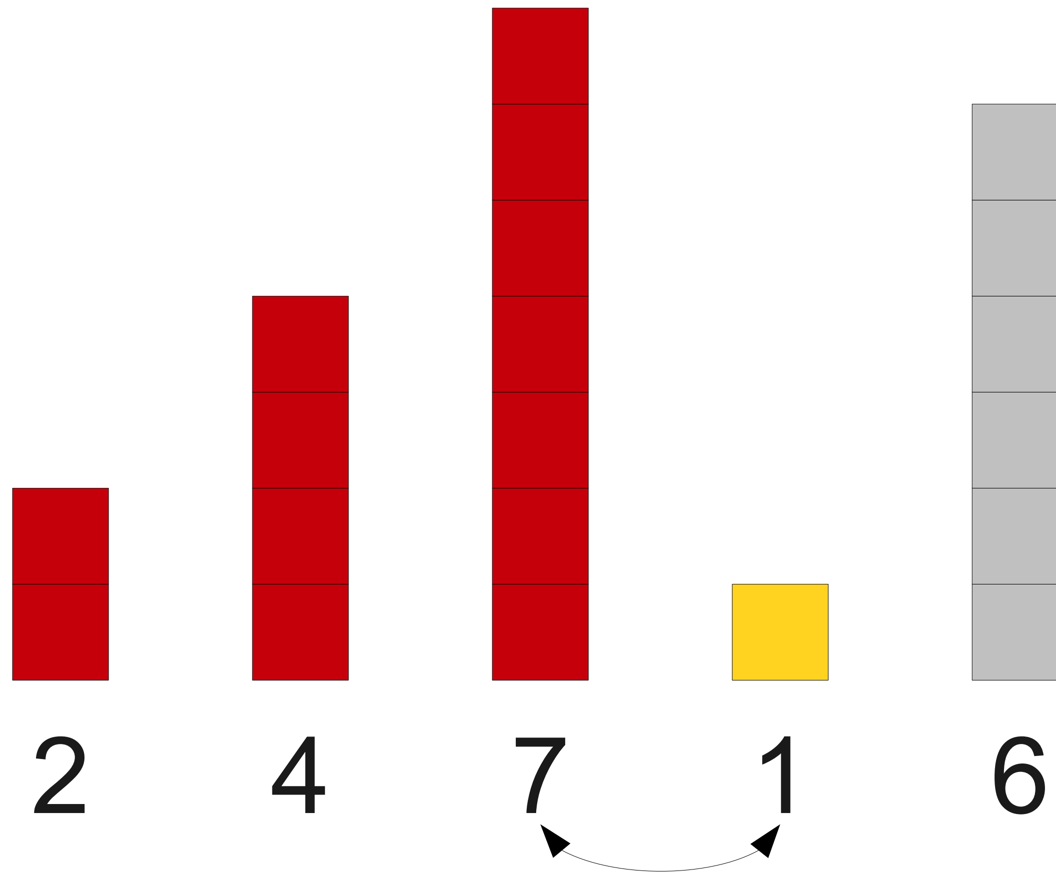
Another Idea: **Insertion Sort**



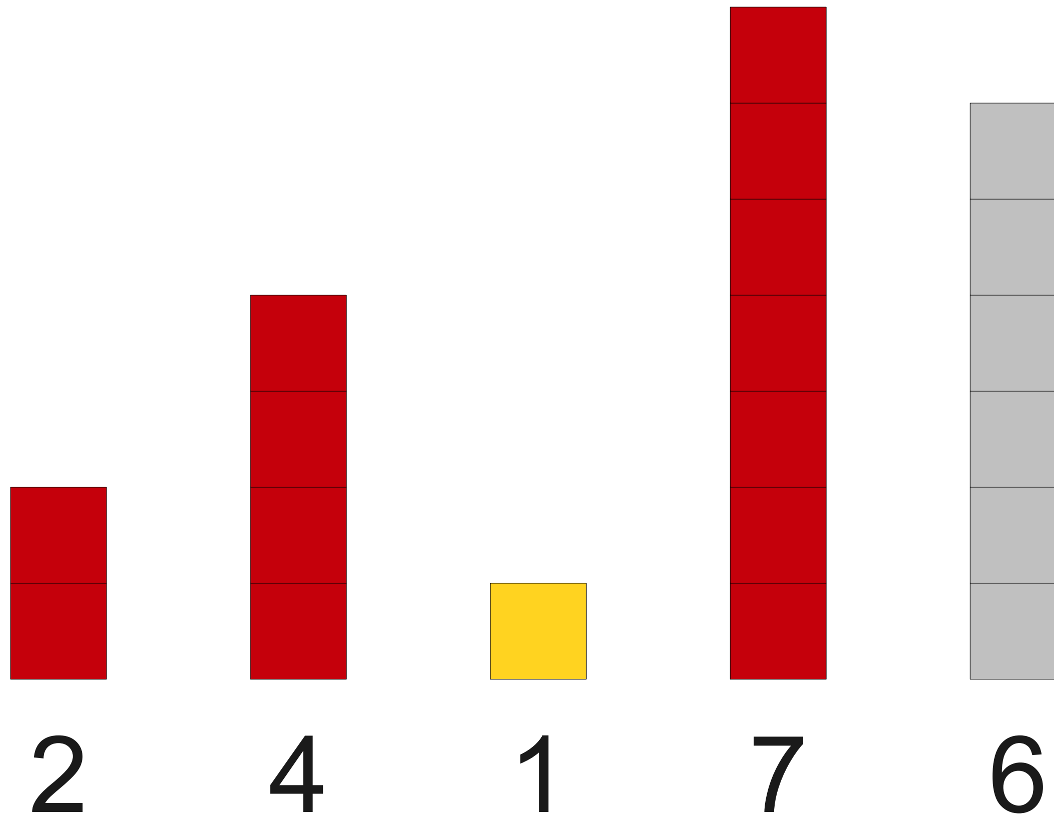
Another Idea: **Insertion Sort**



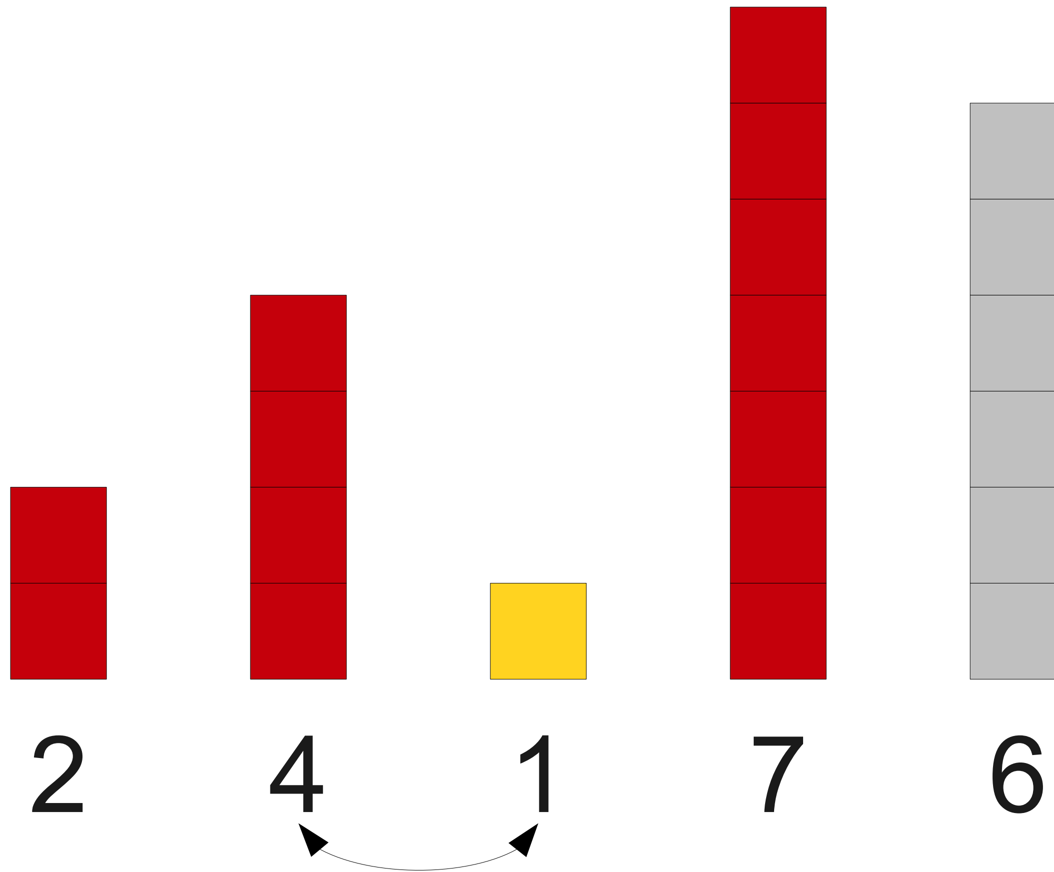
Another Idea: **Insertion Sort**



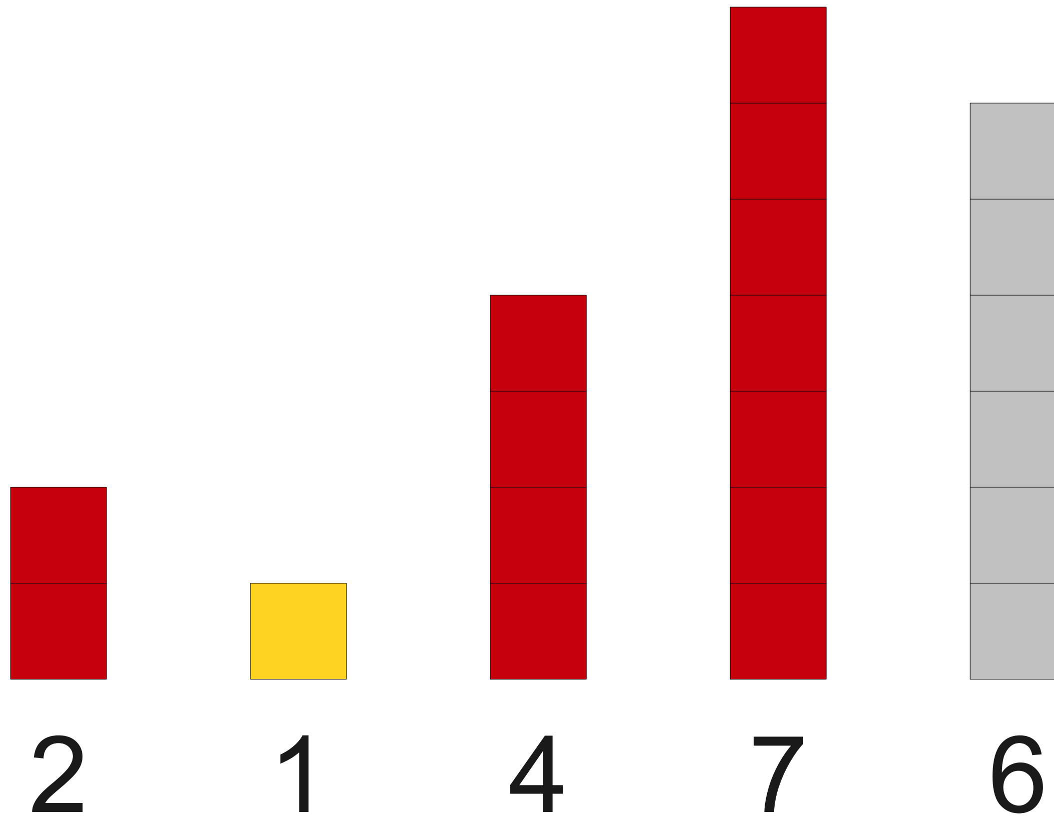
Another Idea: **Insertion Sort**



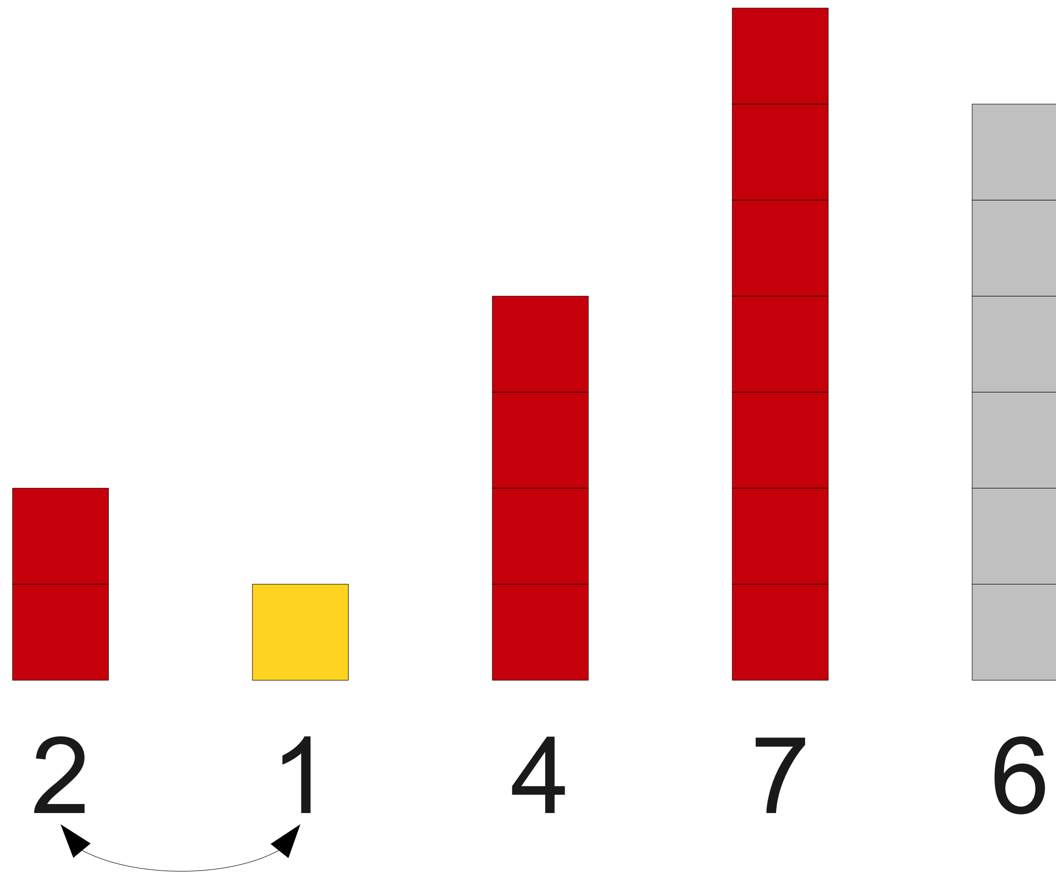
Another Idea: **Insertion Sort**



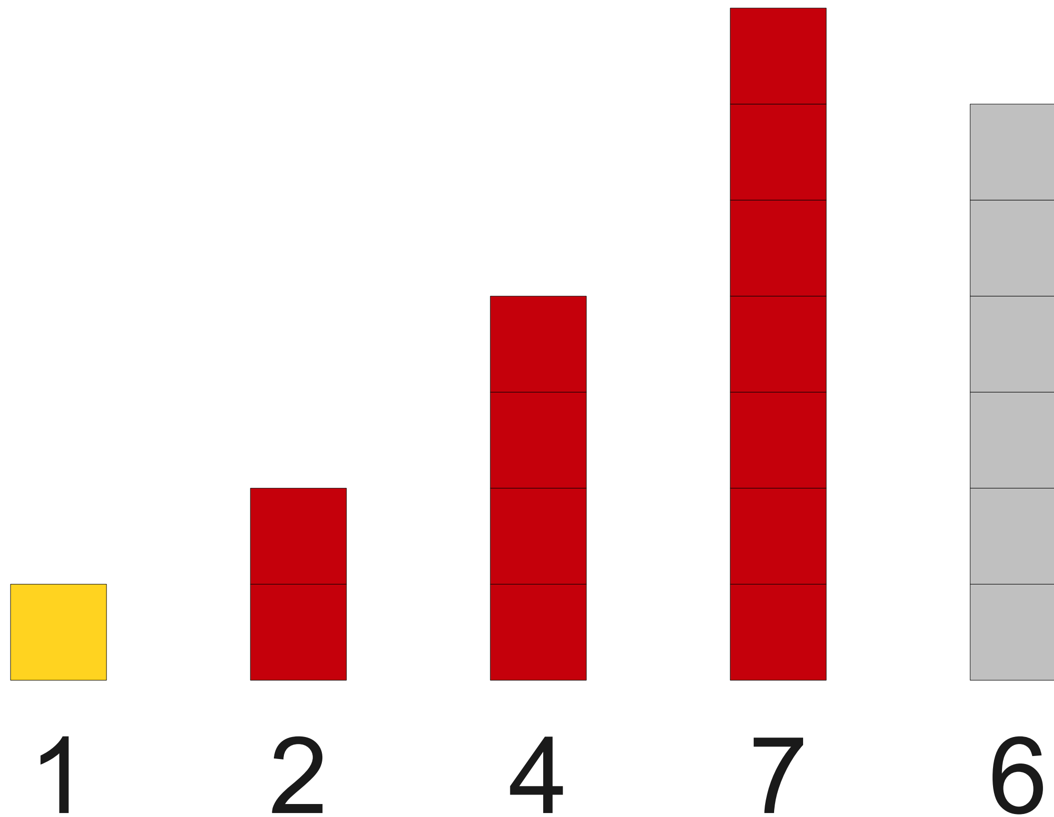
Another Idea: **Insertion Sort**



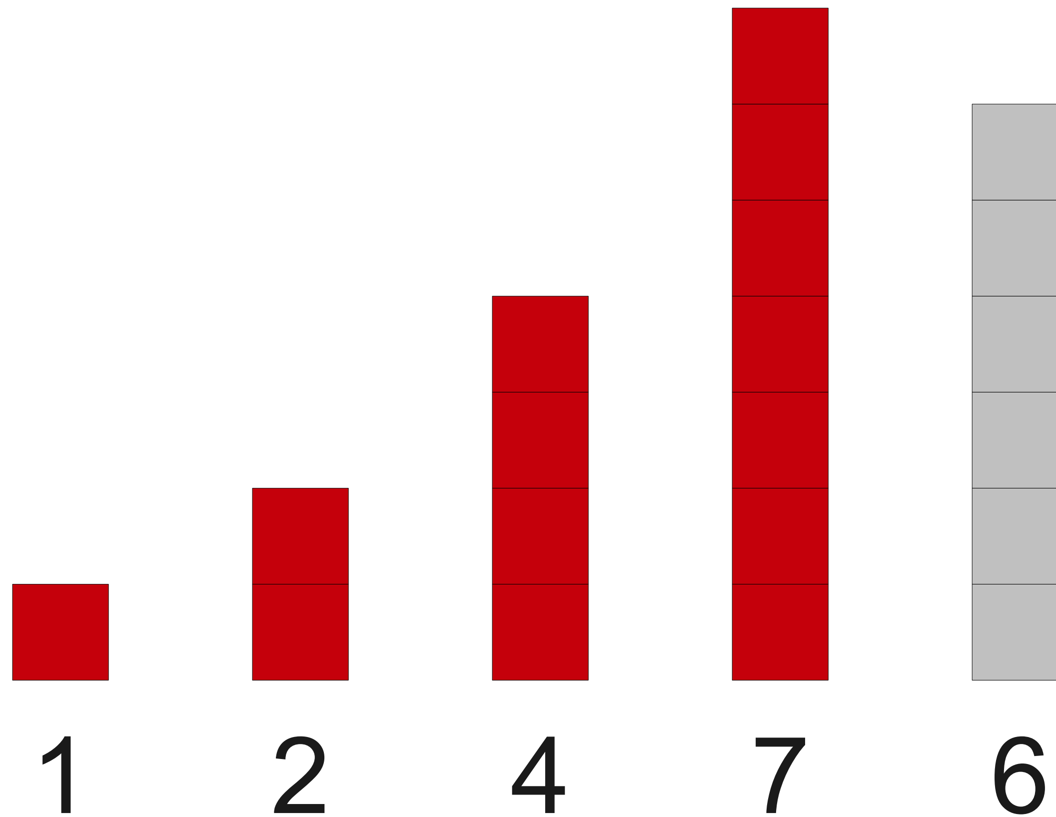
Another Idea: **Insertion Sort**



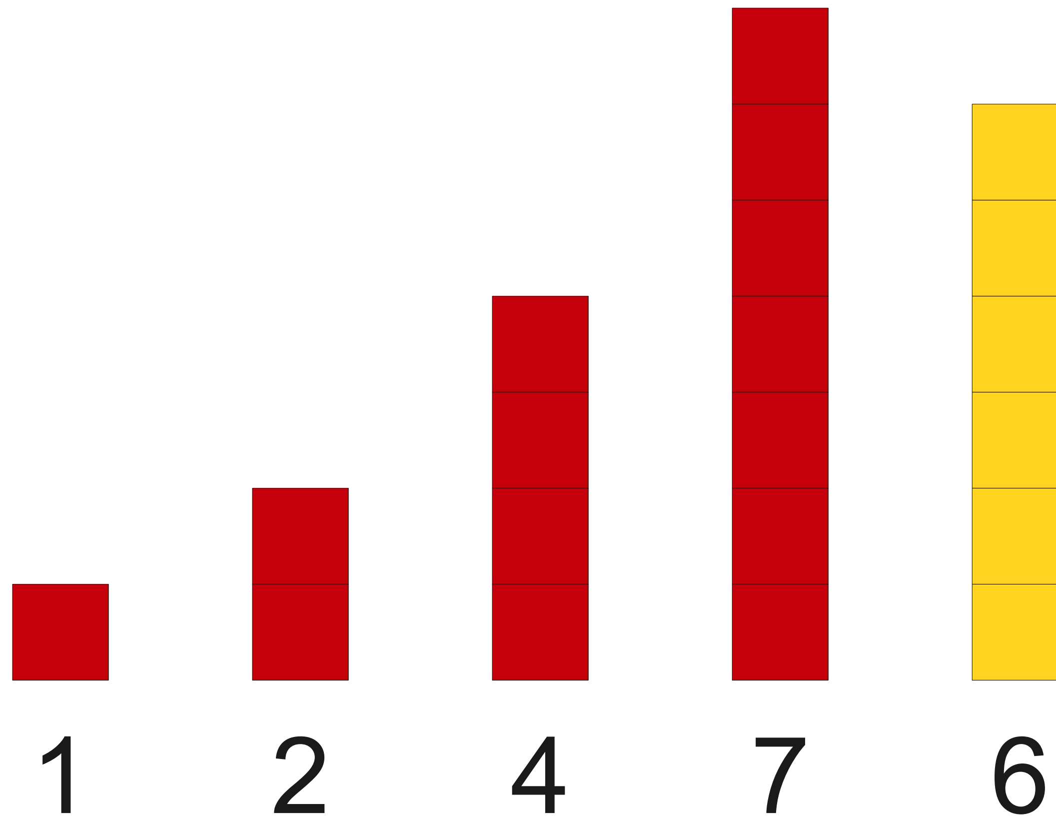
Another Idea: **Insertion Sort**



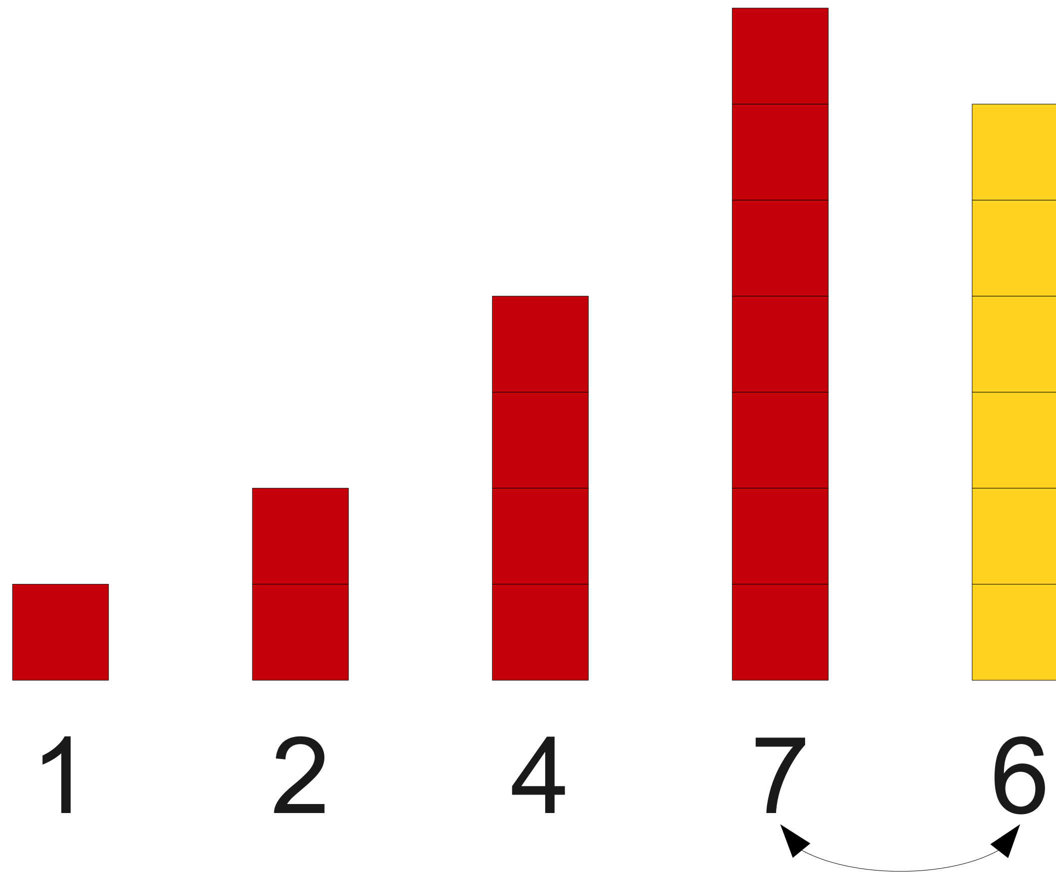
Another Idea: **Insertion Sort**



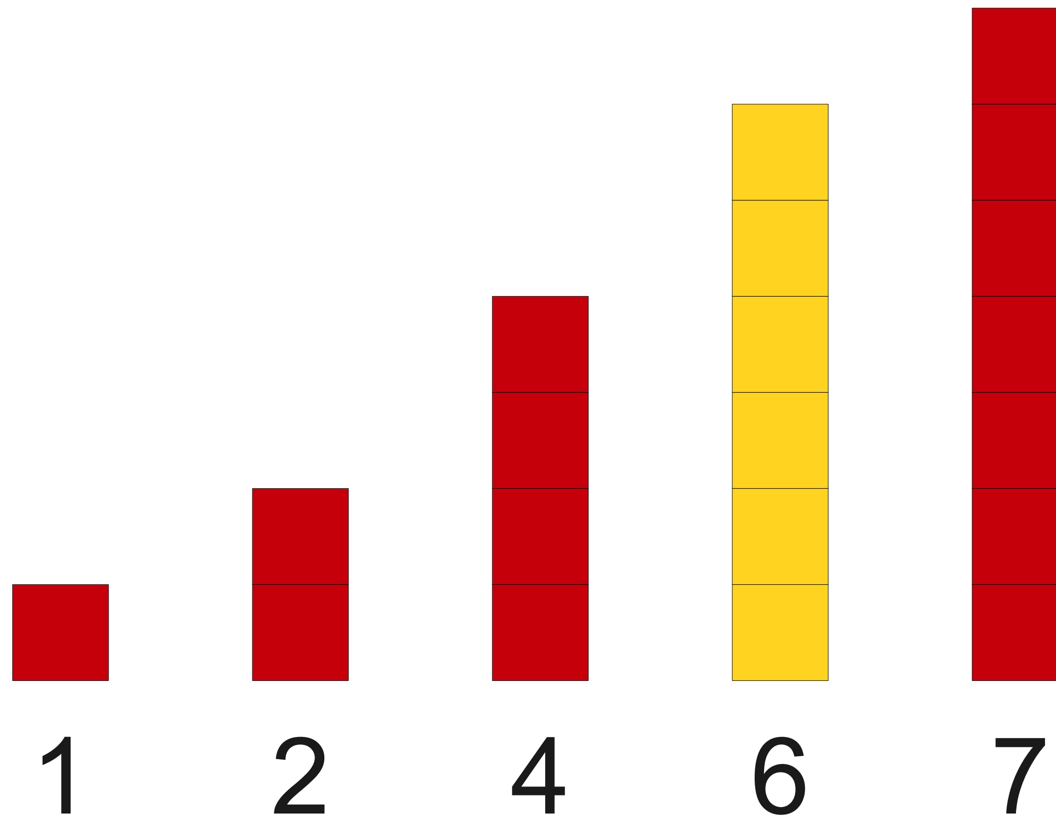
Another Idea: **Insertion Sort**



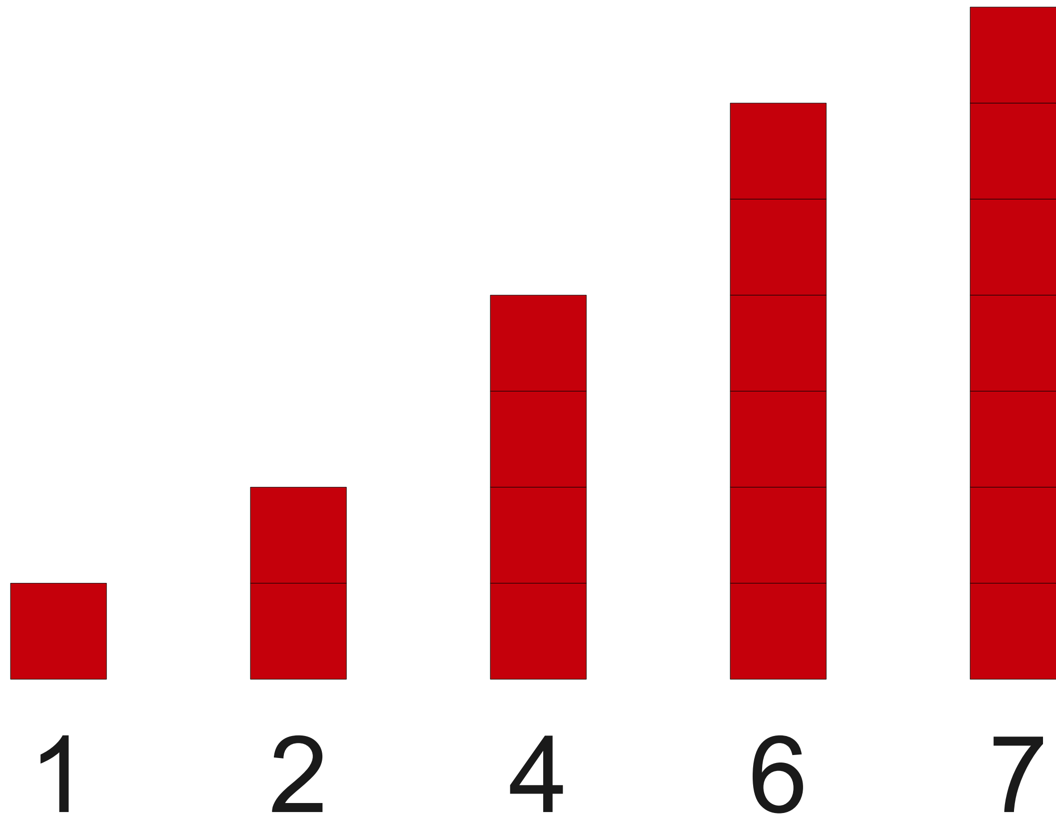
Another Idea: **Insertion Sort**



Another Idea: **Insertion Sort**



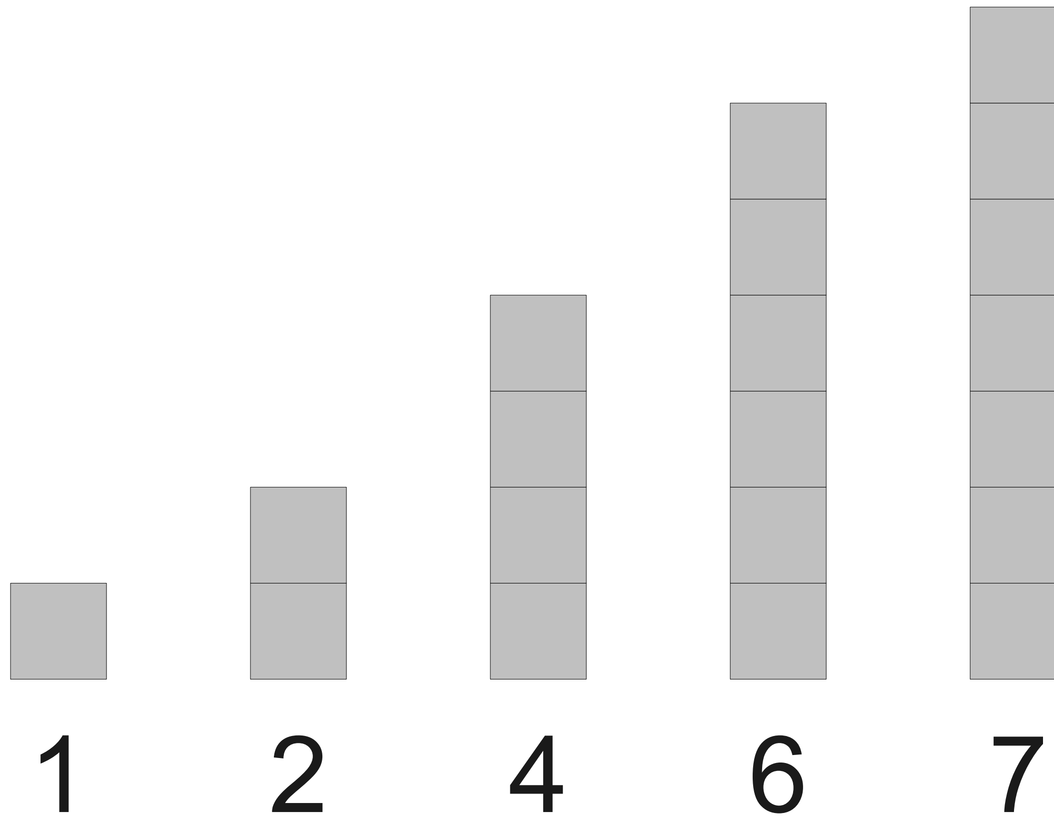
Another Idea: **Insertion Sort**



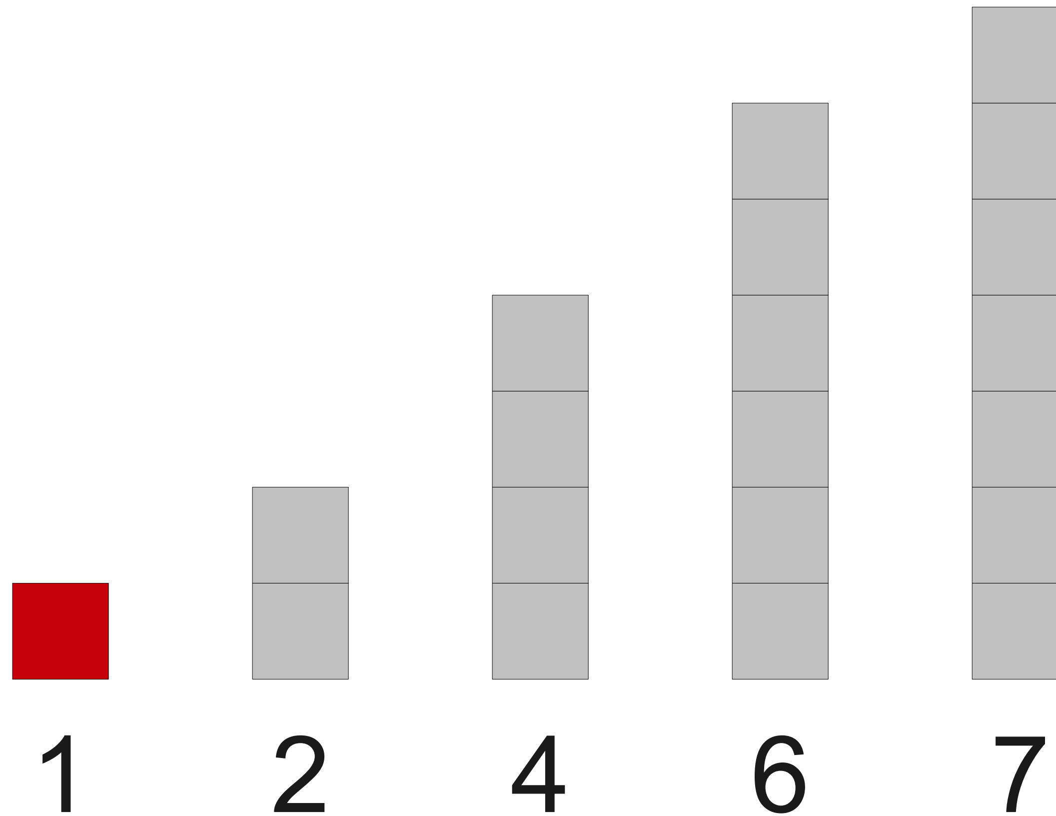
Insertion Sort in Code

```
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]);
        }
    }
}
```

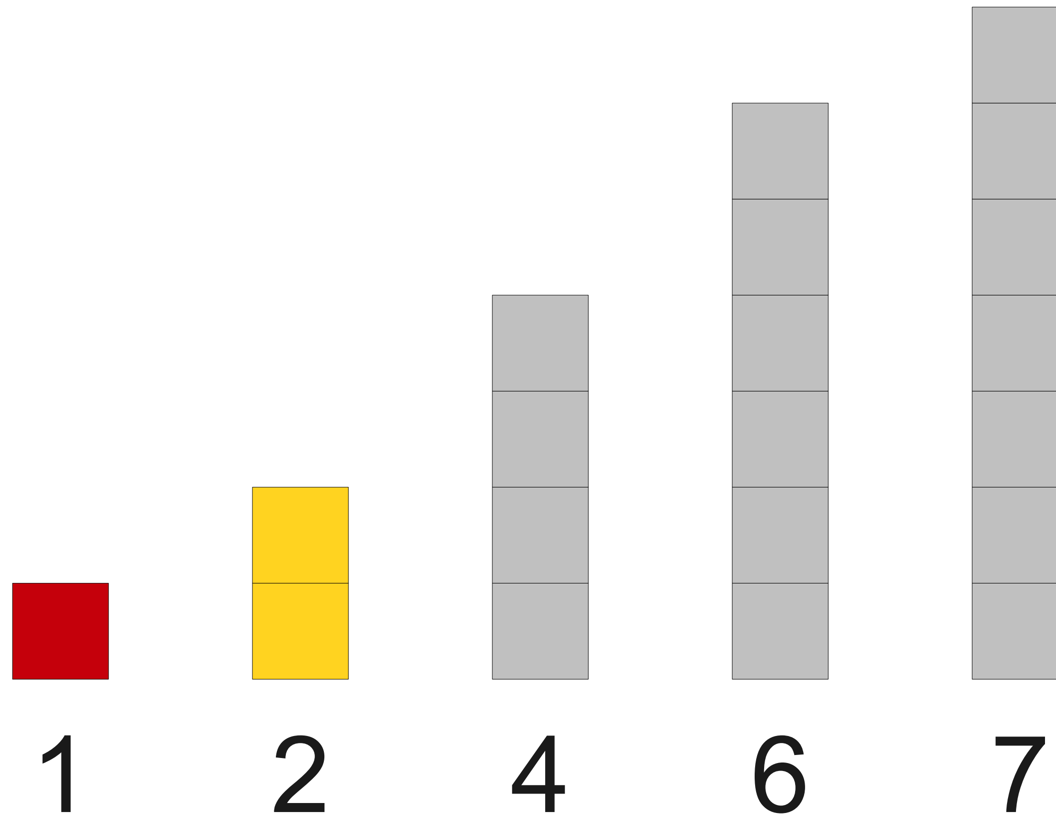
How Fast is Insertion Sort?



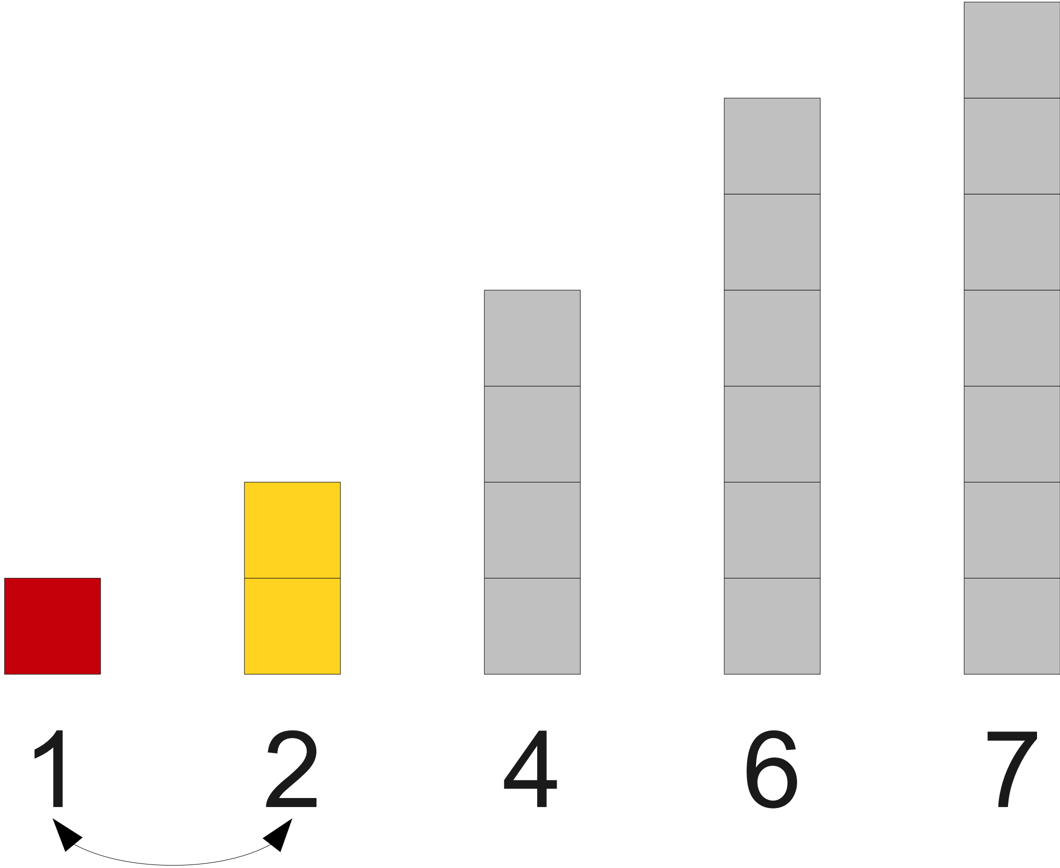
How Fast is Insertion Sort?



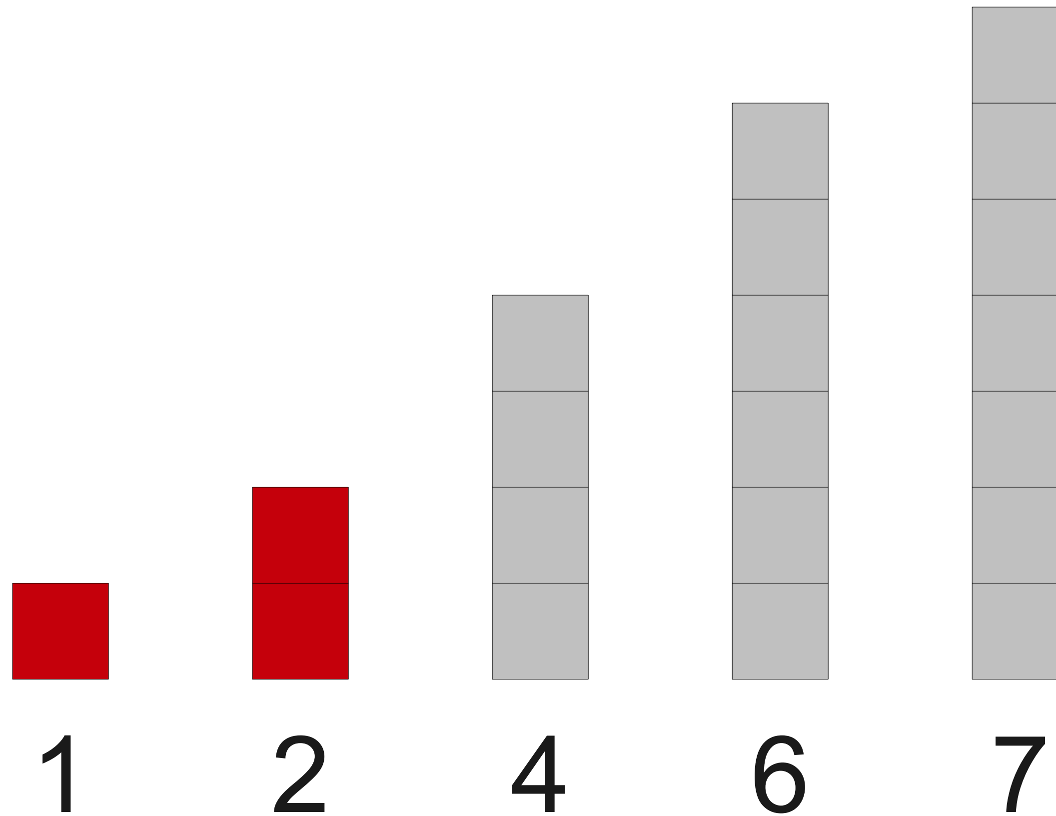
How Fast is Insertion Sort?



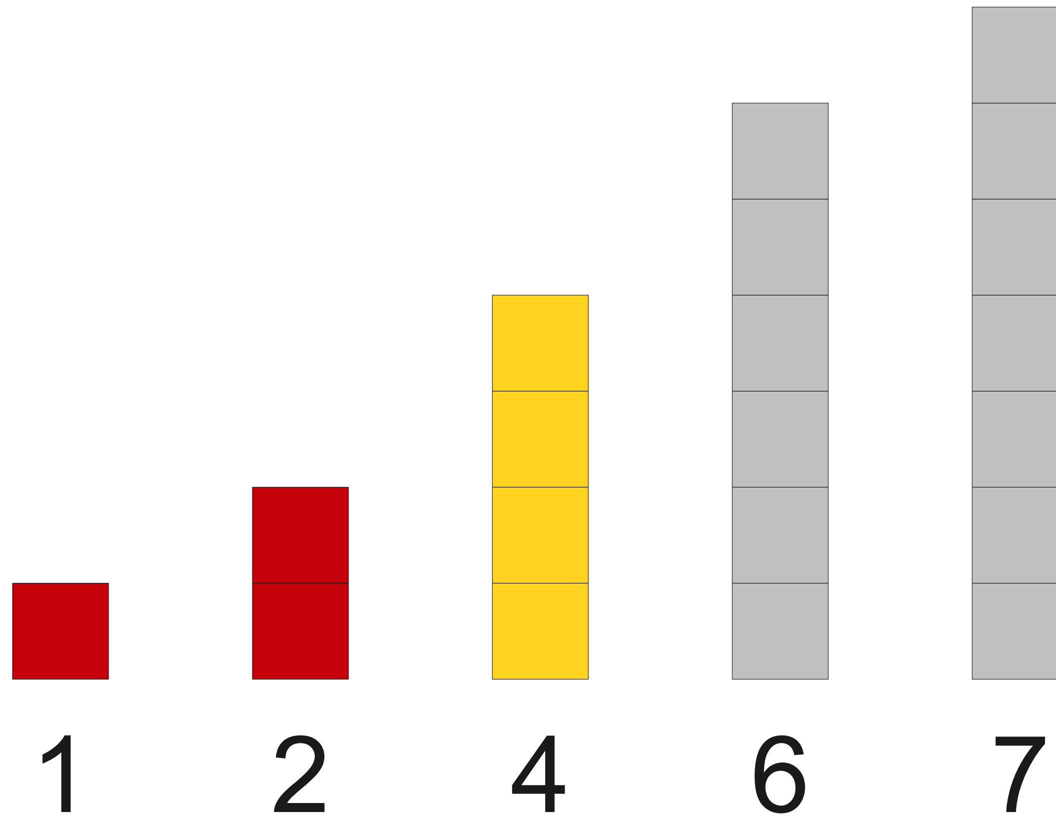
How Fast is Insertion Sort?



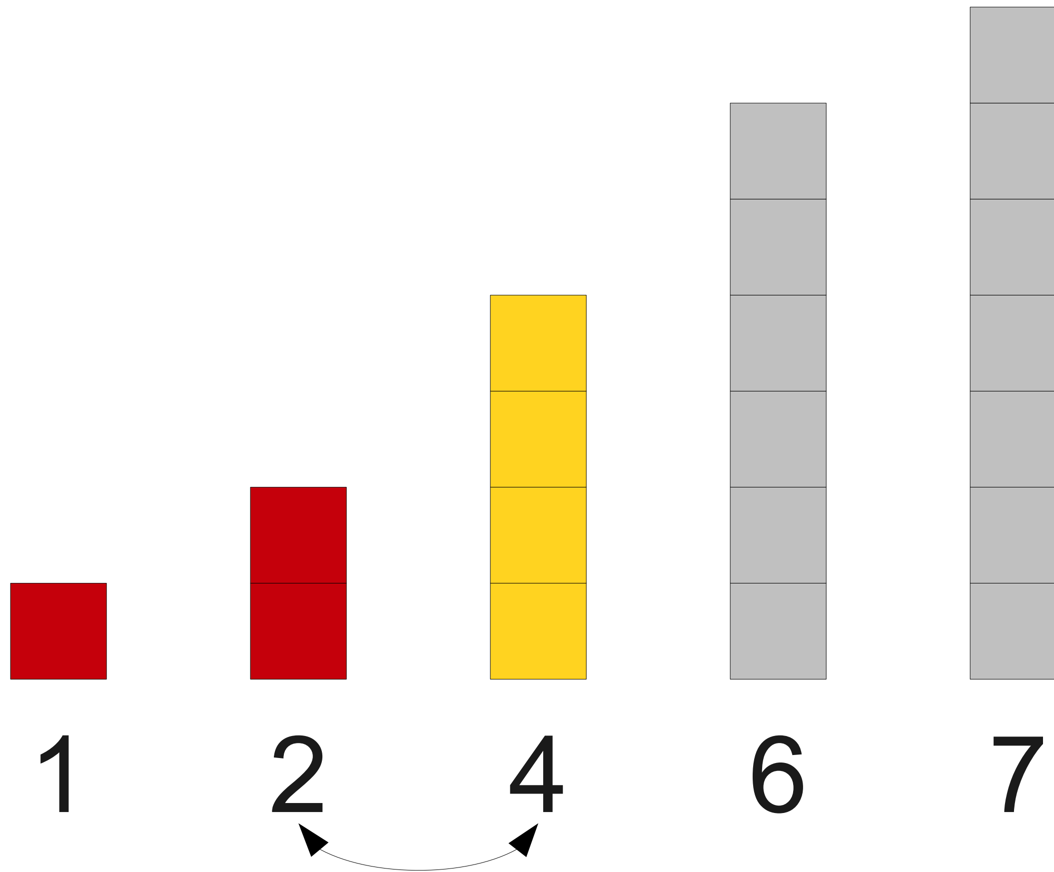
How Fast is Insertion Sort?



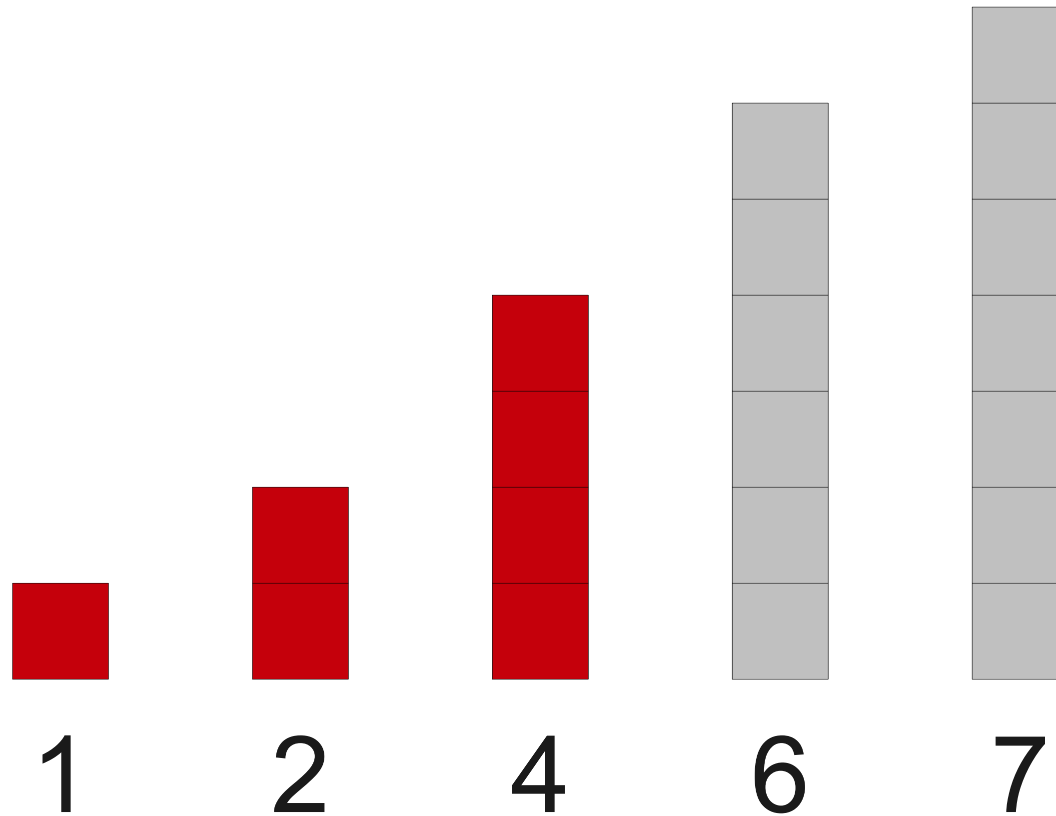
How Fast is Insertion Sort?



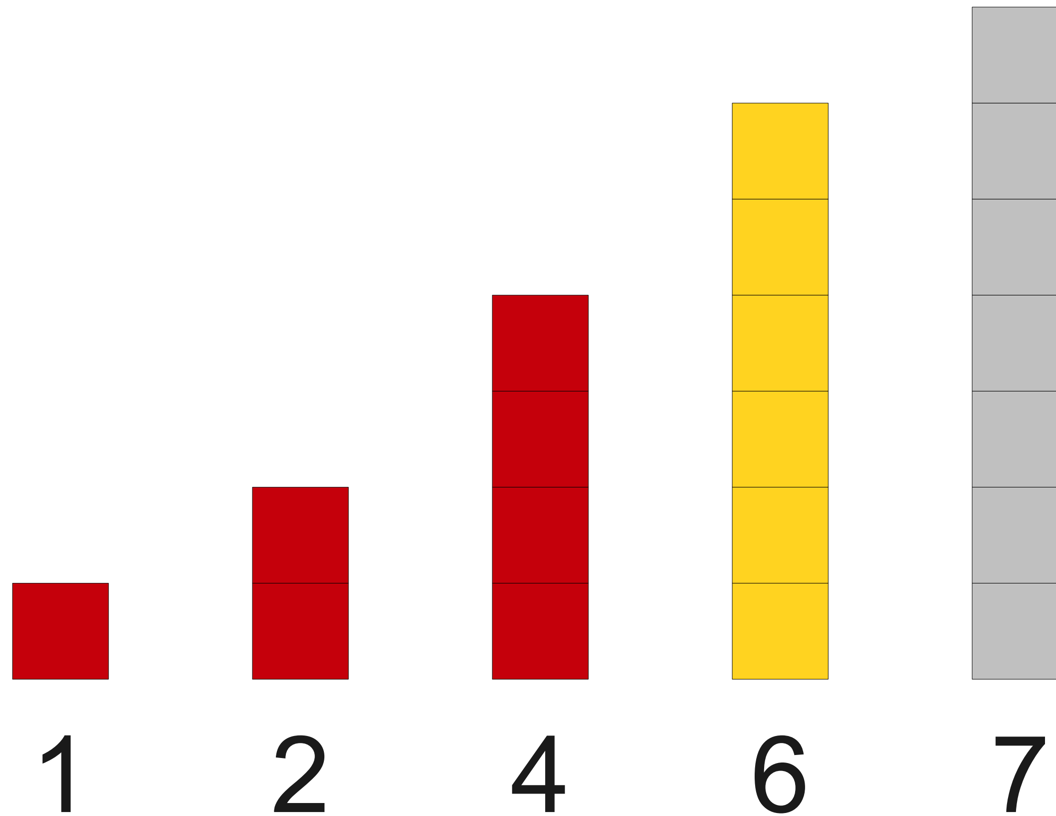
How Fast is Insertion Sort?



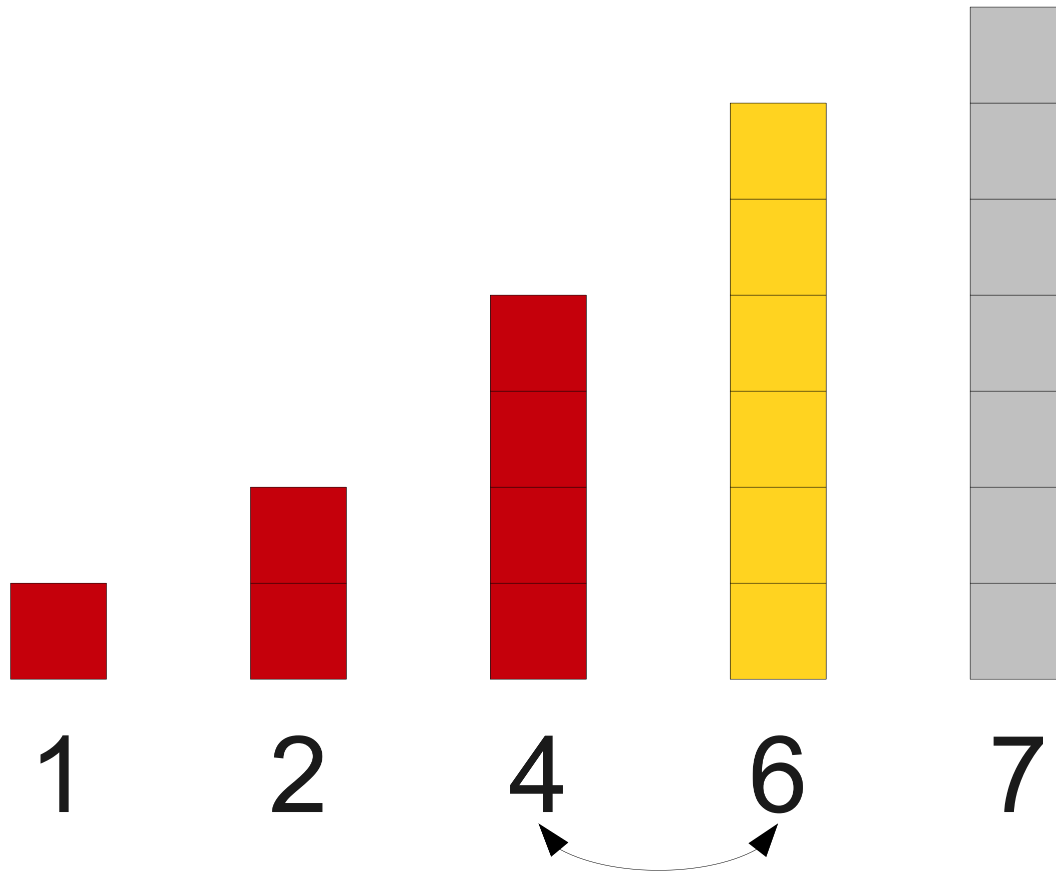
How Fast is Insertion Sort?



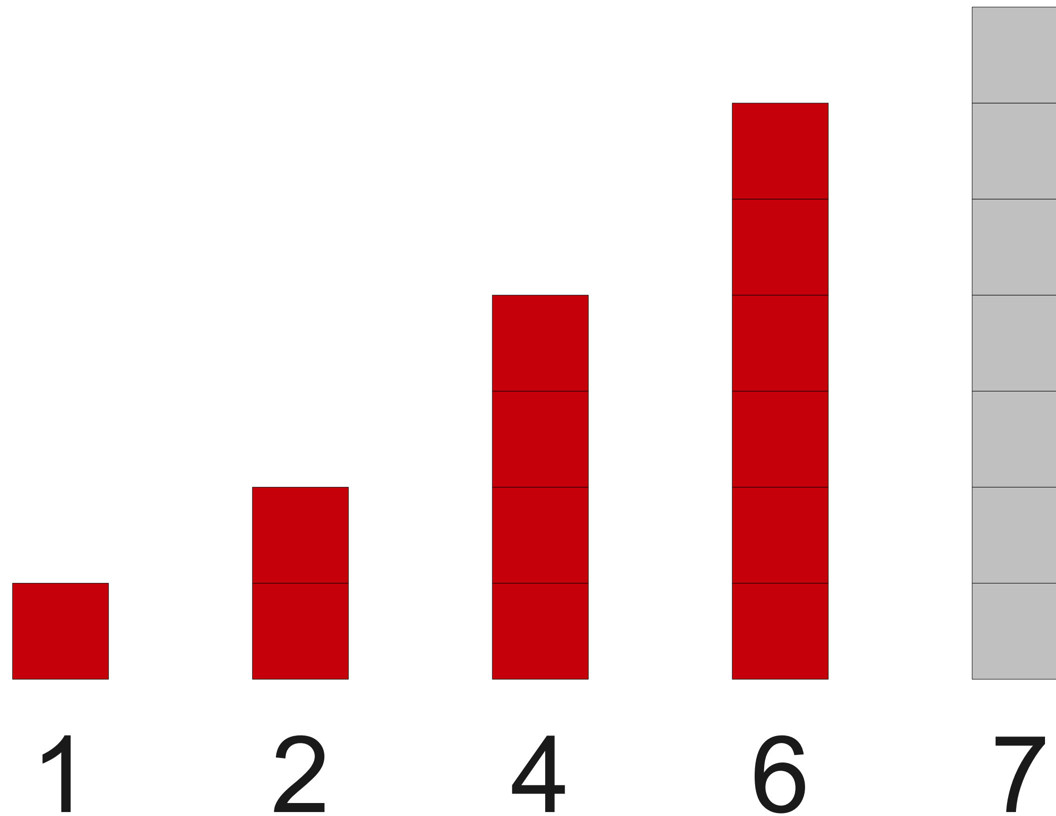
How Fast is Insertion Sort?



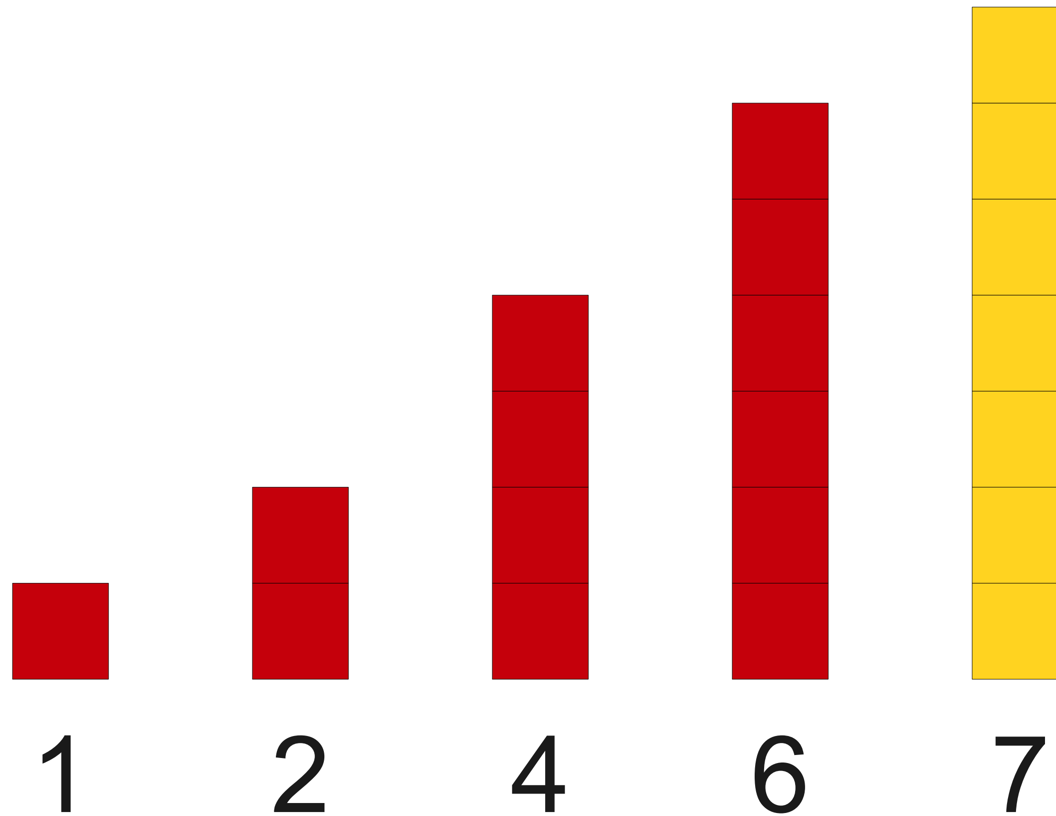
How Fast is Insertion Sort?



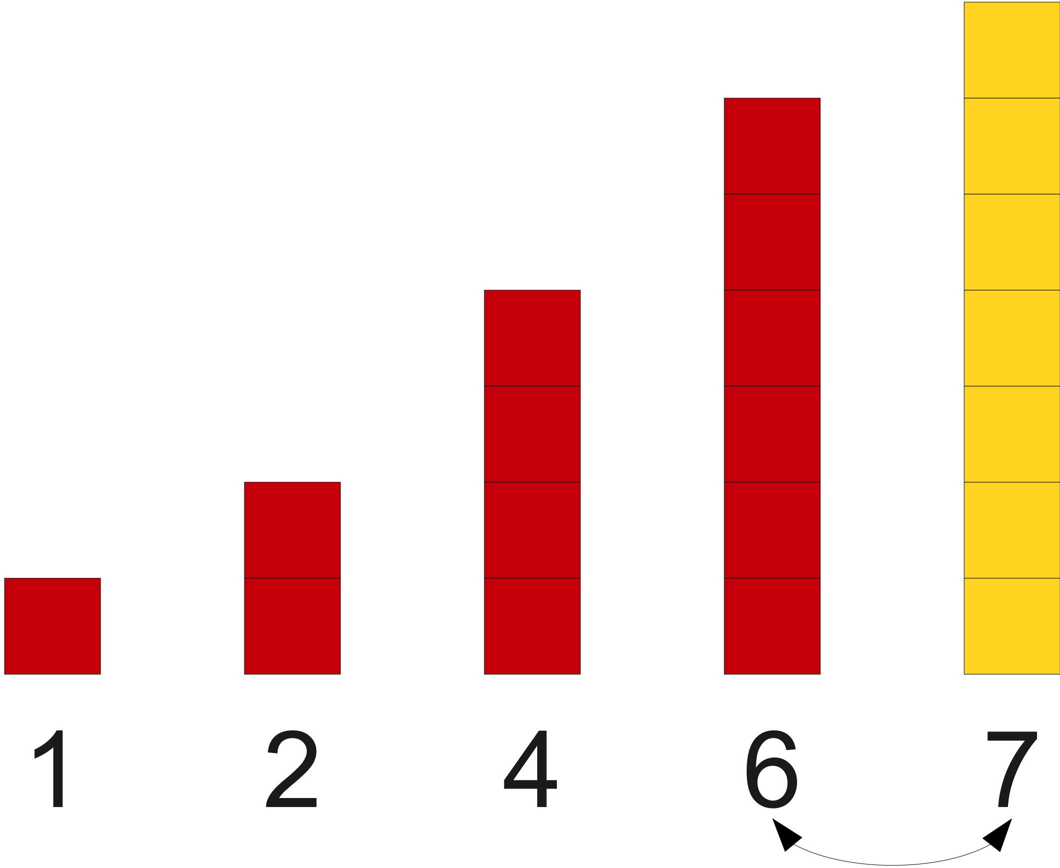
How Fast is Insertion Sort?



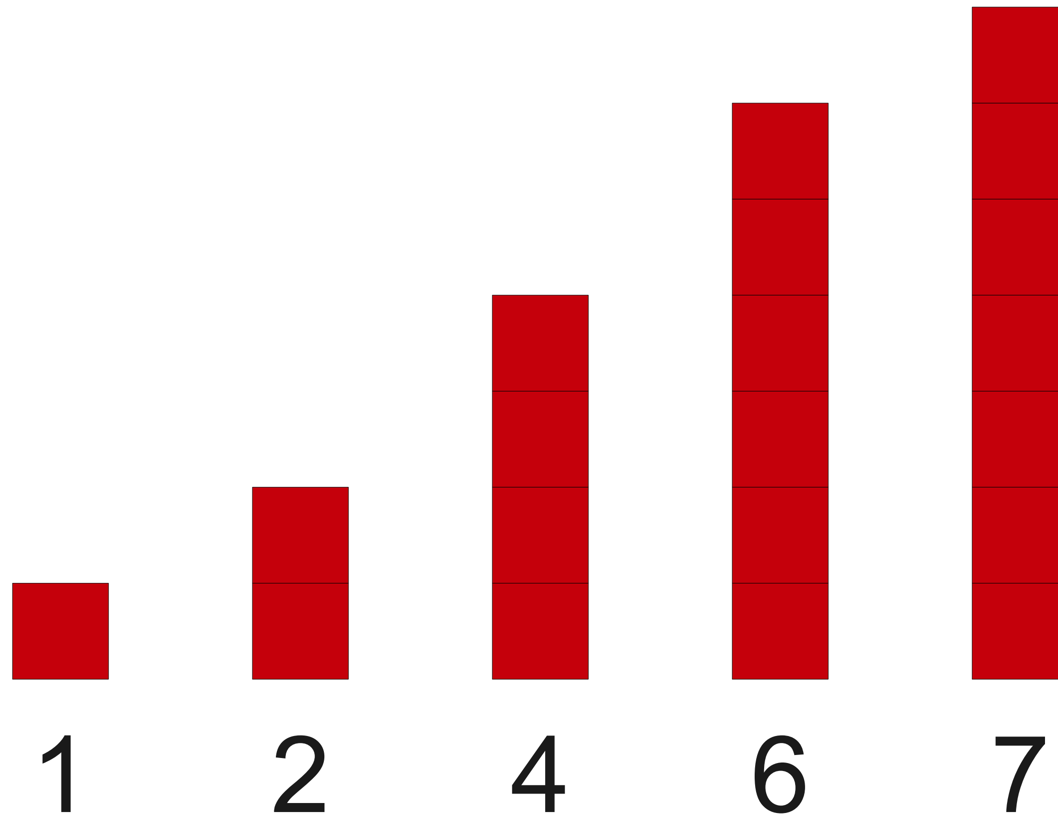
How Fast is Insertion Sort?



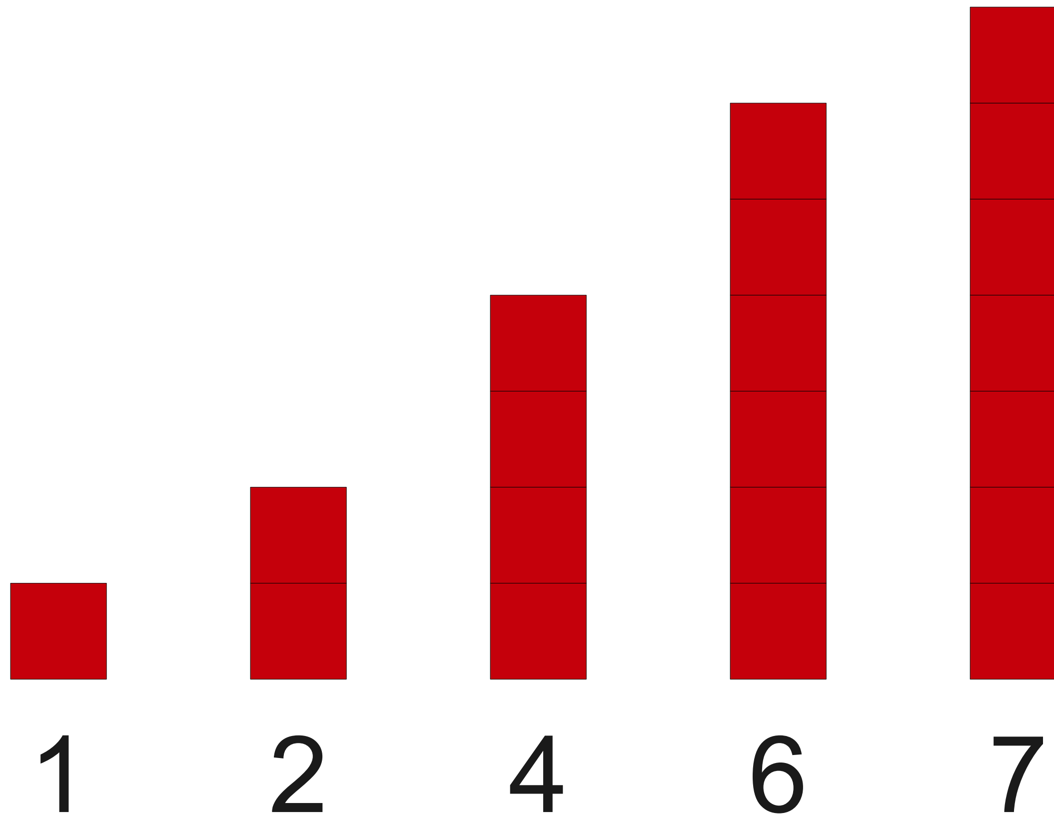
How Fast is Insertion Sort?



How Fast is Insertion Sort?

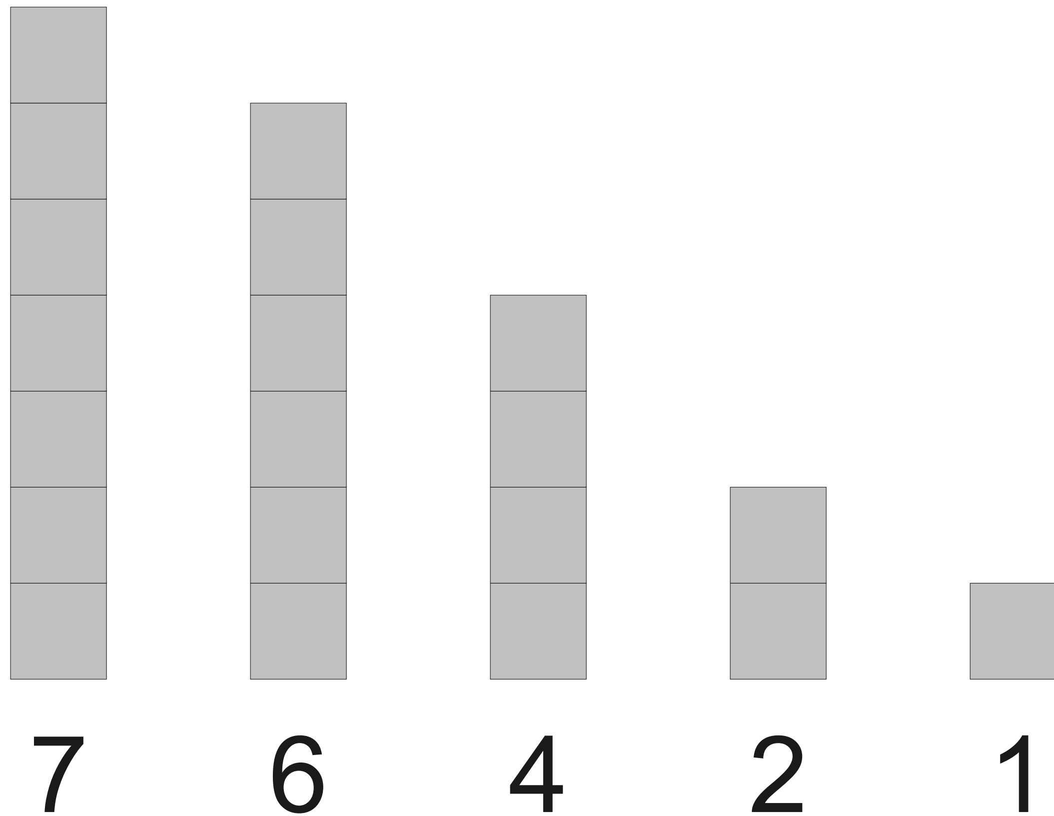


How Fast is Insertion Sort?

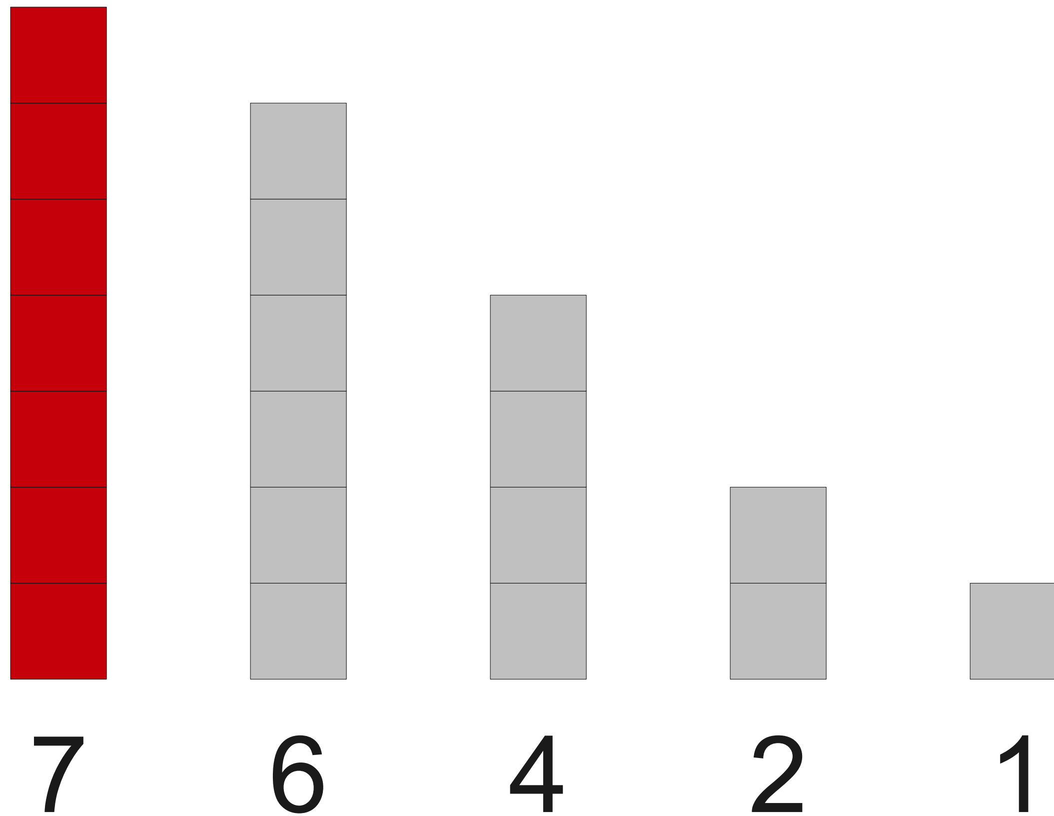


Work done: **O(n)**

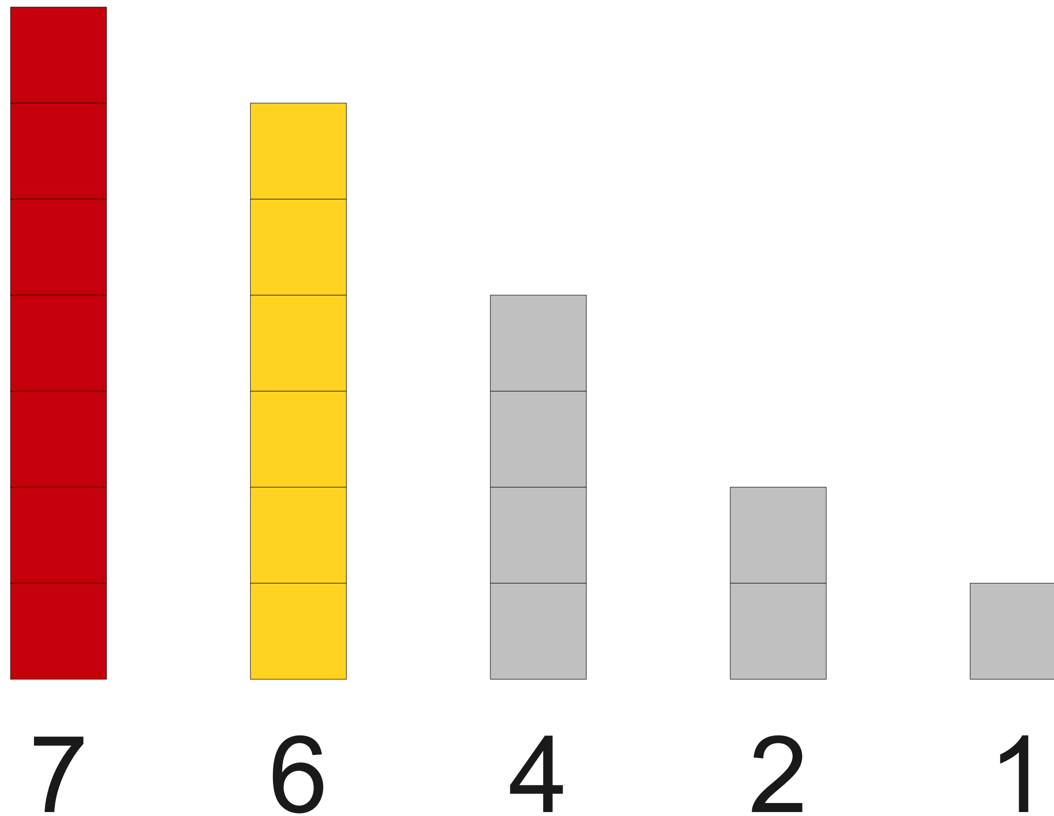
How Fast is Insertion Sort?



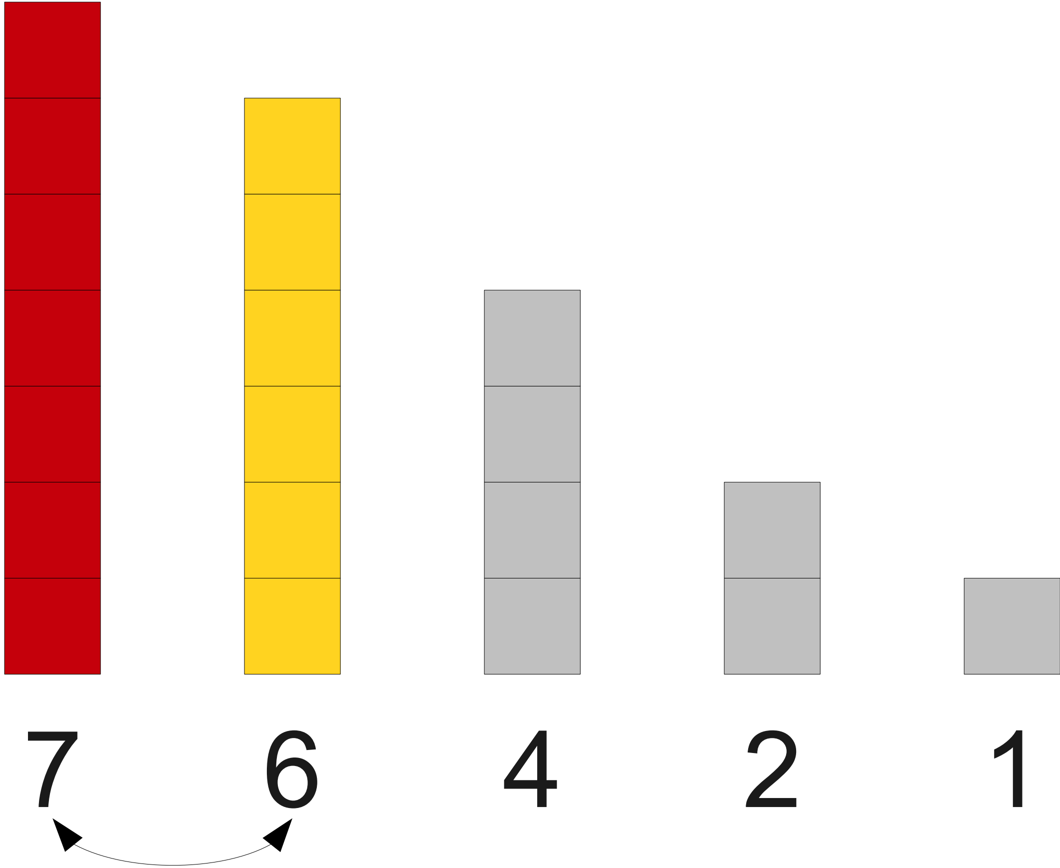
How Fast is Insertion Sort?



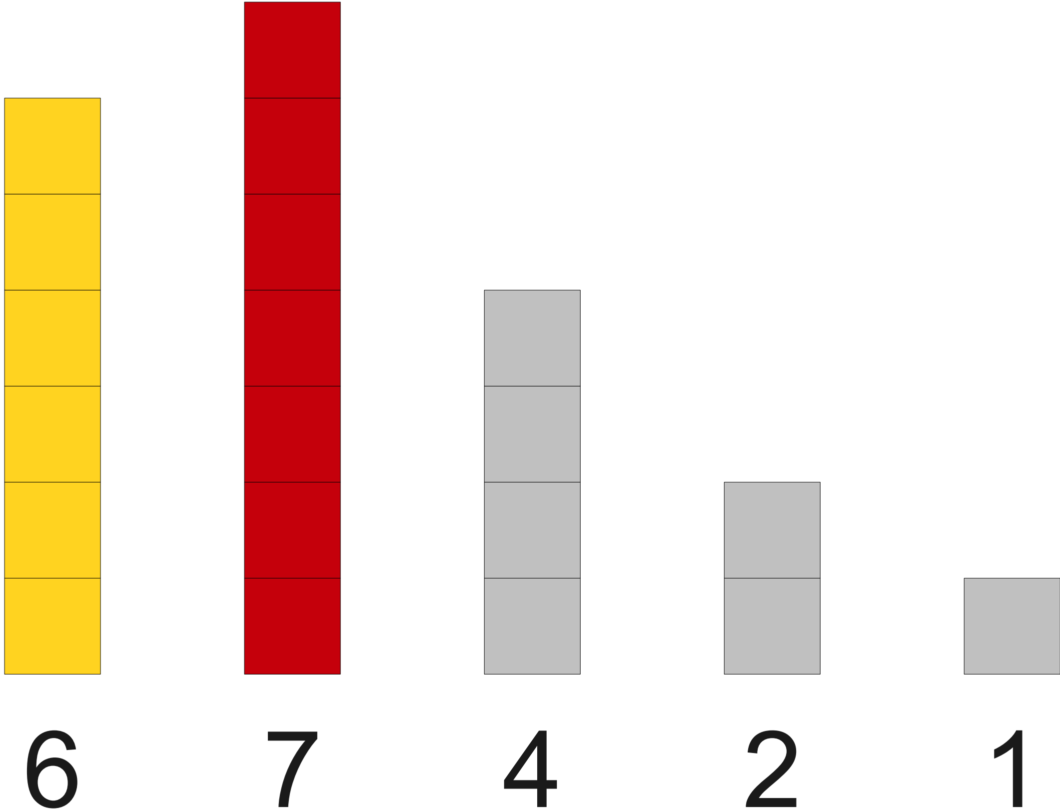
How Fast is Insertion Sort?



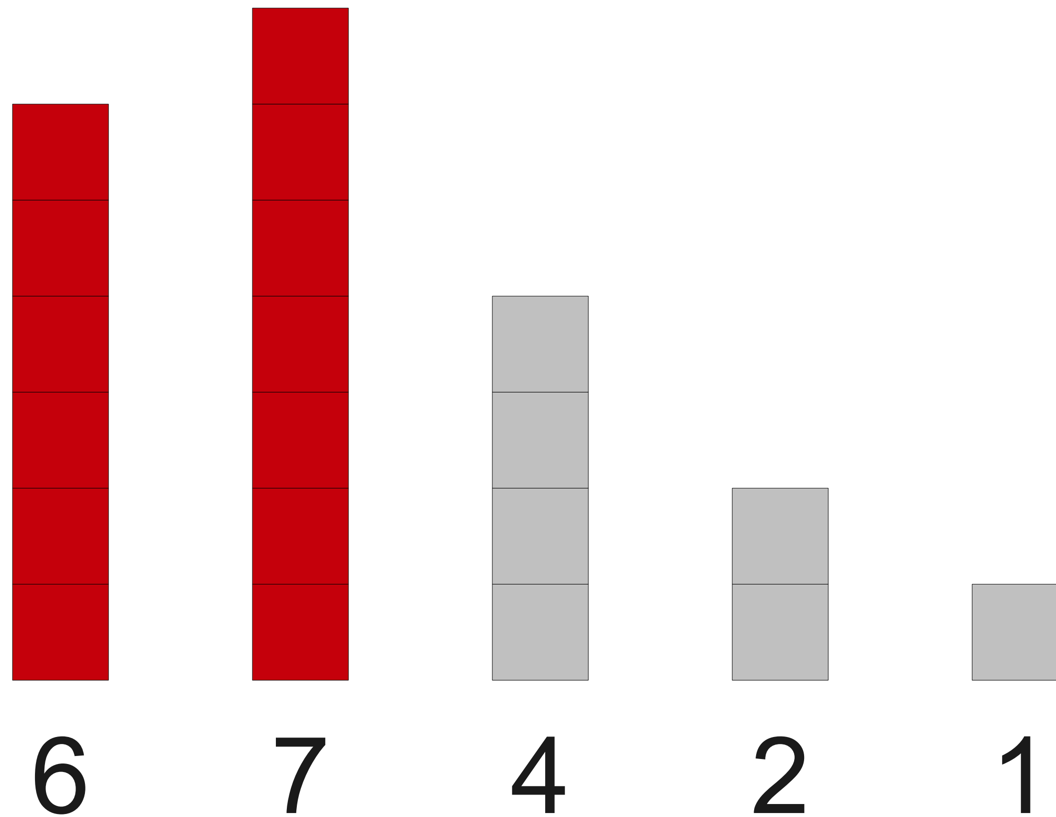
How Fast is Insertion Sort?



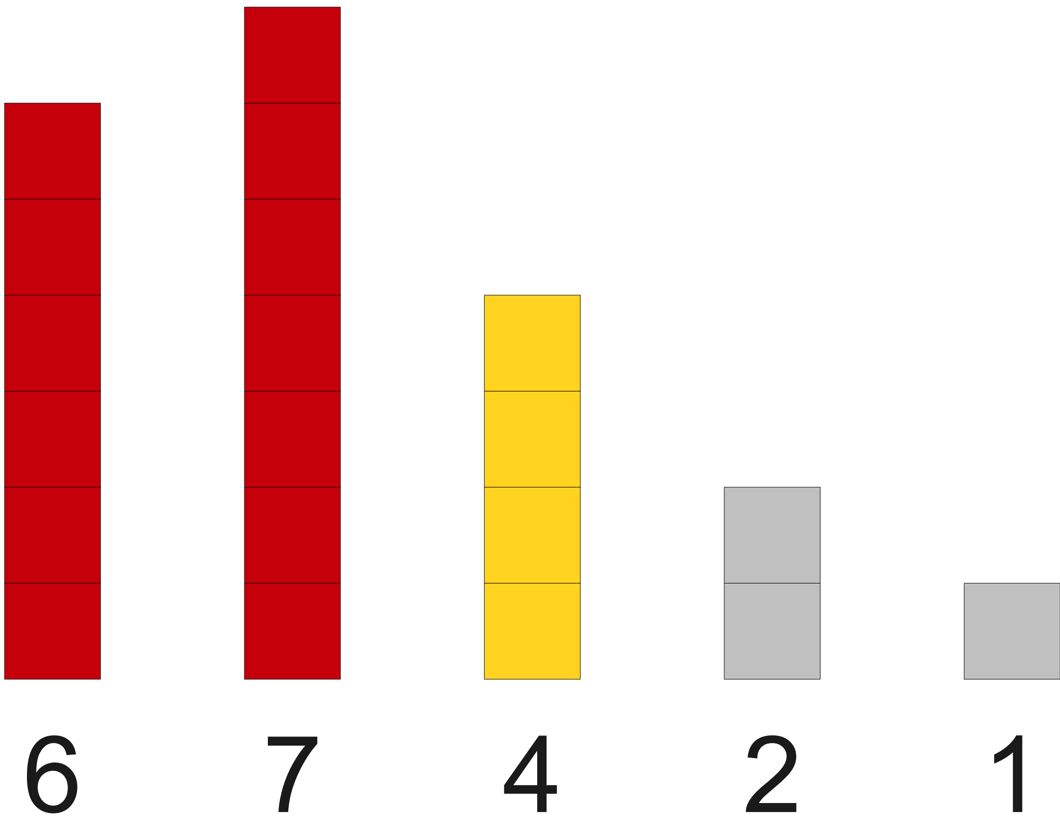
How Fast is Insertion Sort?



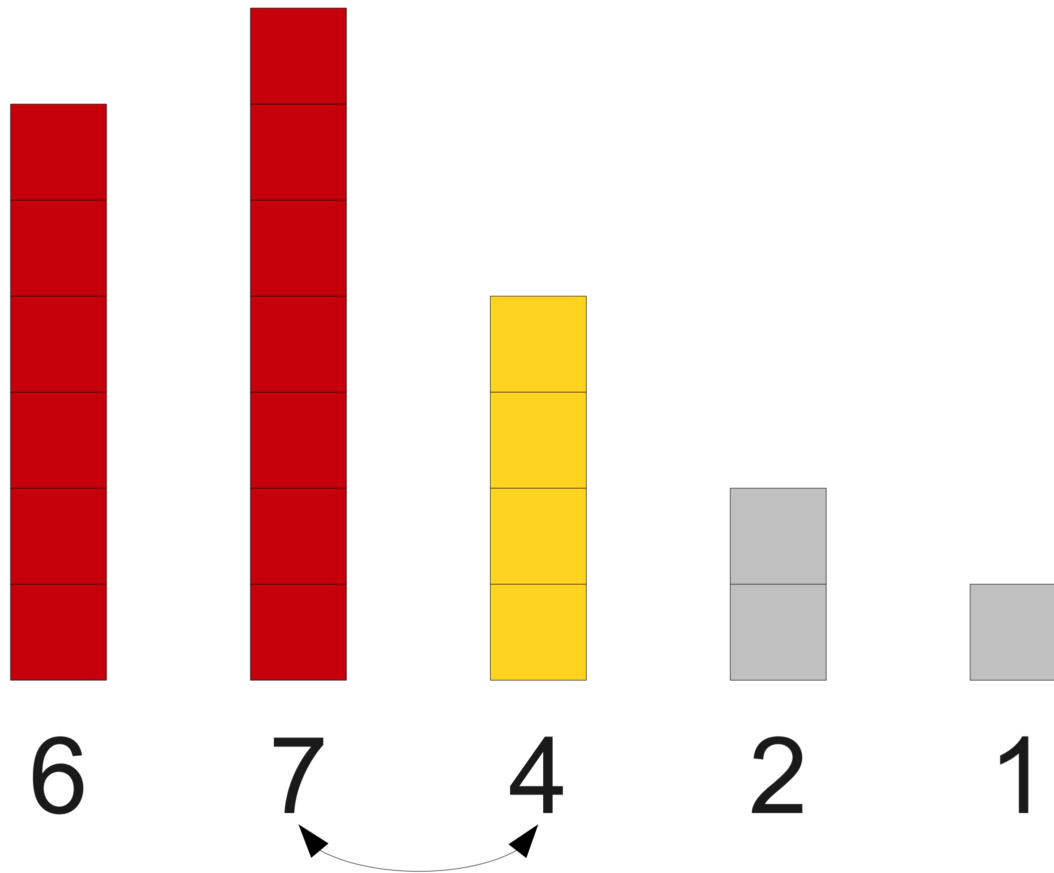
How Fast is Insertion Sort?



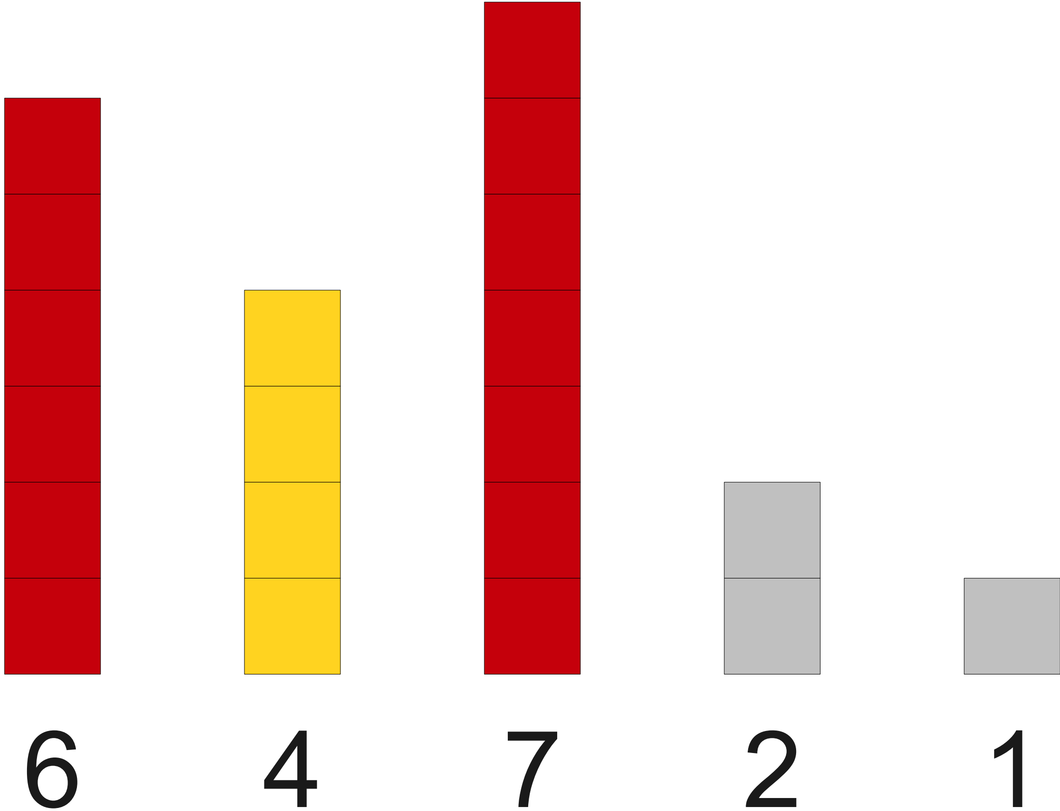
How Fast is Insertion Sort?



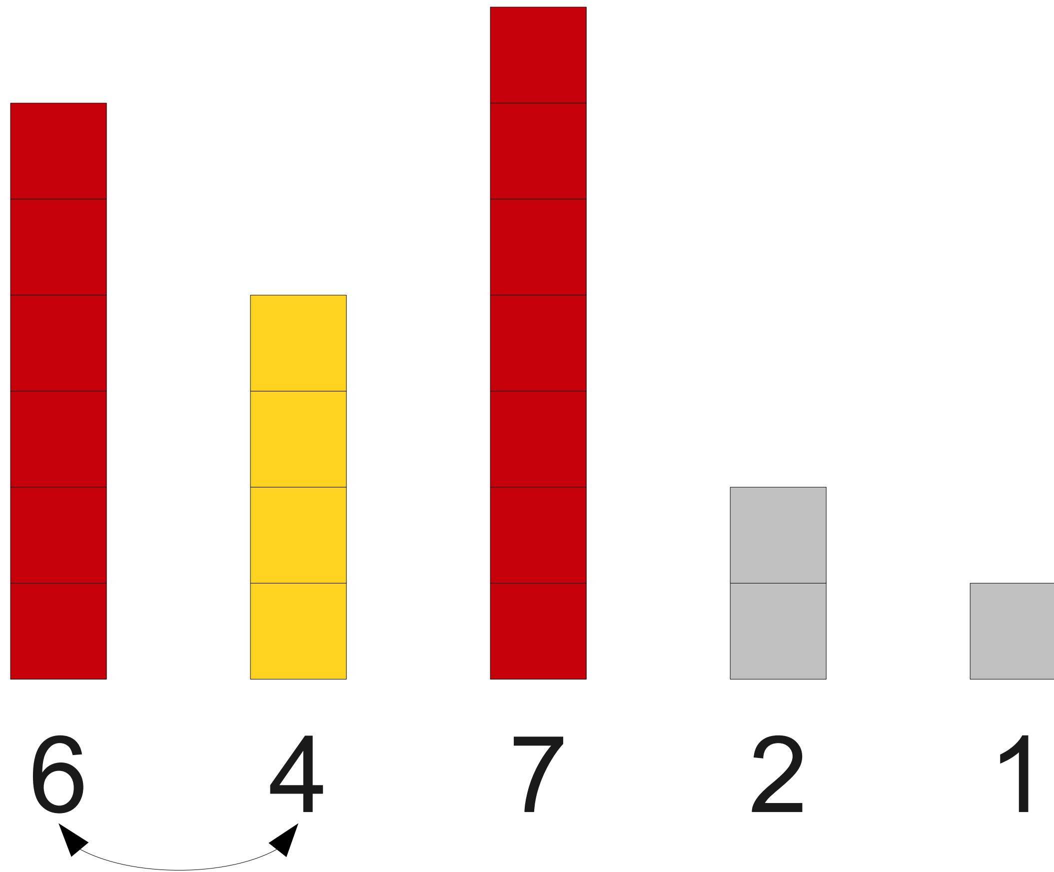
How Fast is Insertion Sort?



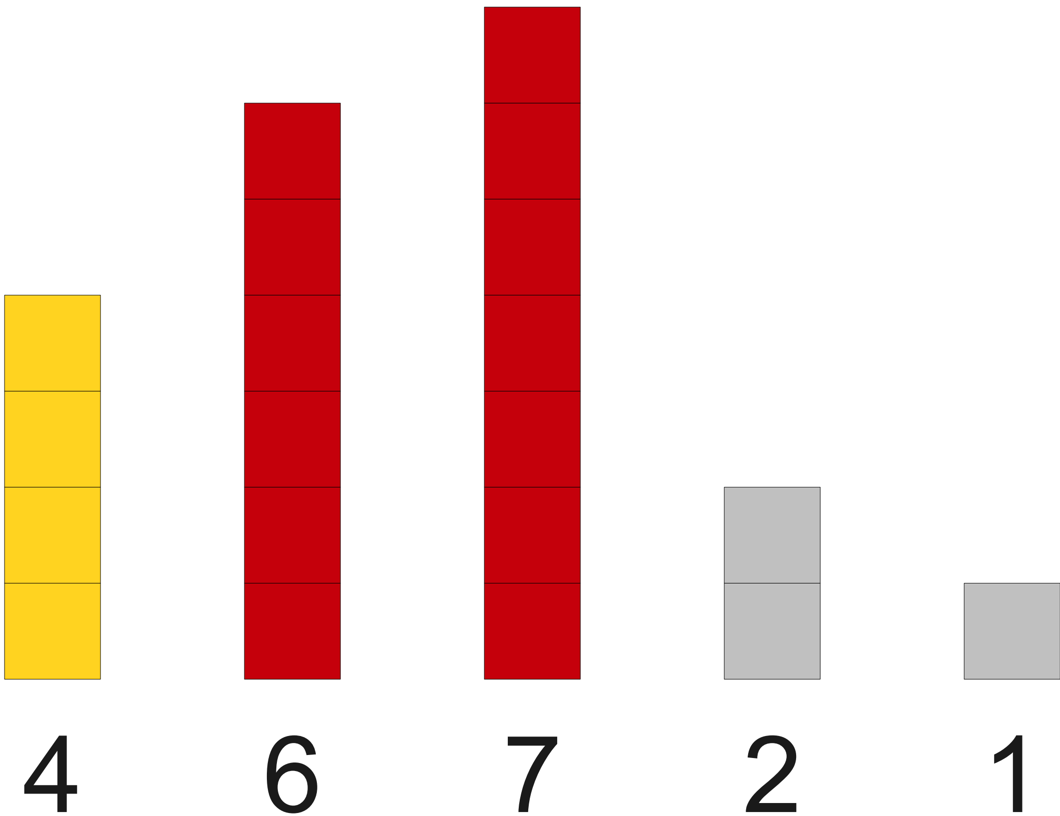
How Fast is Insertion Sort?



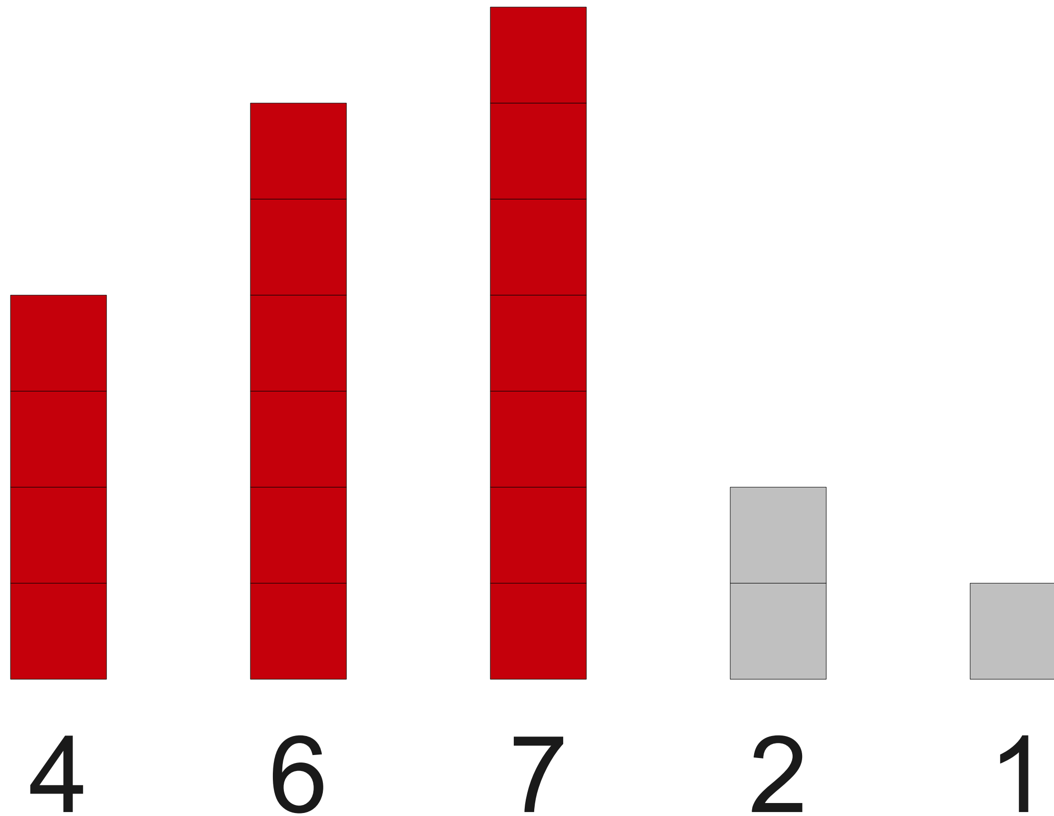
How Fast is Insertion Sort?



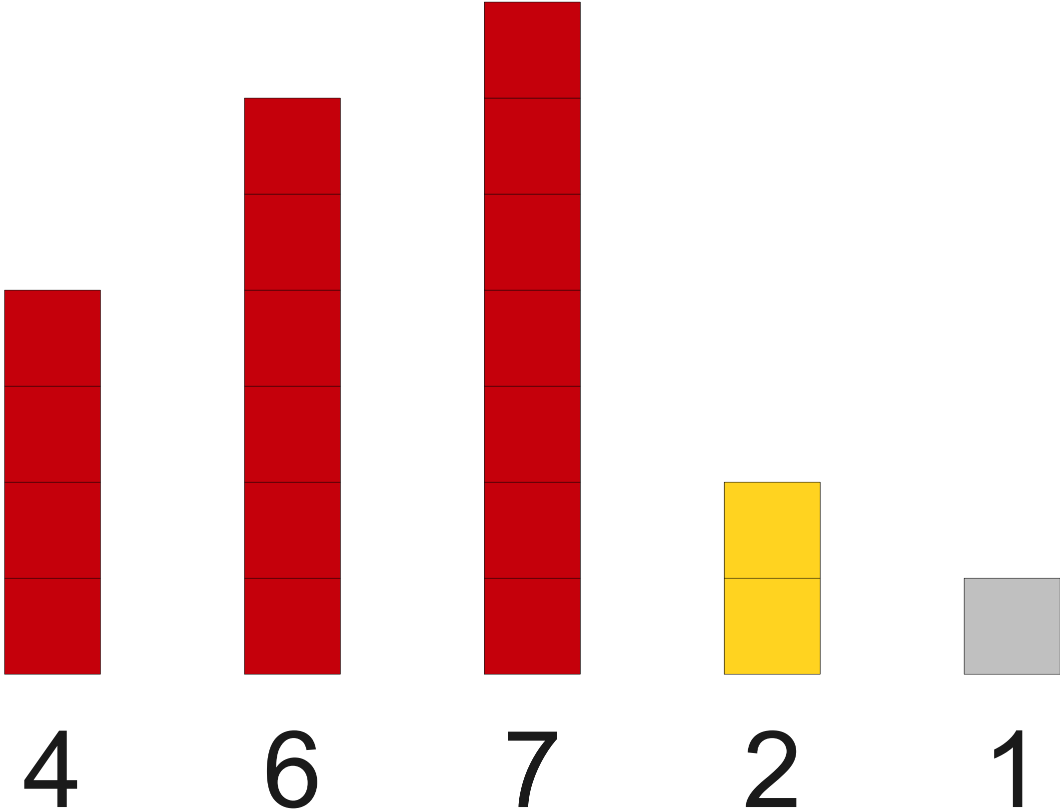
How Fast is Insertion Sort?



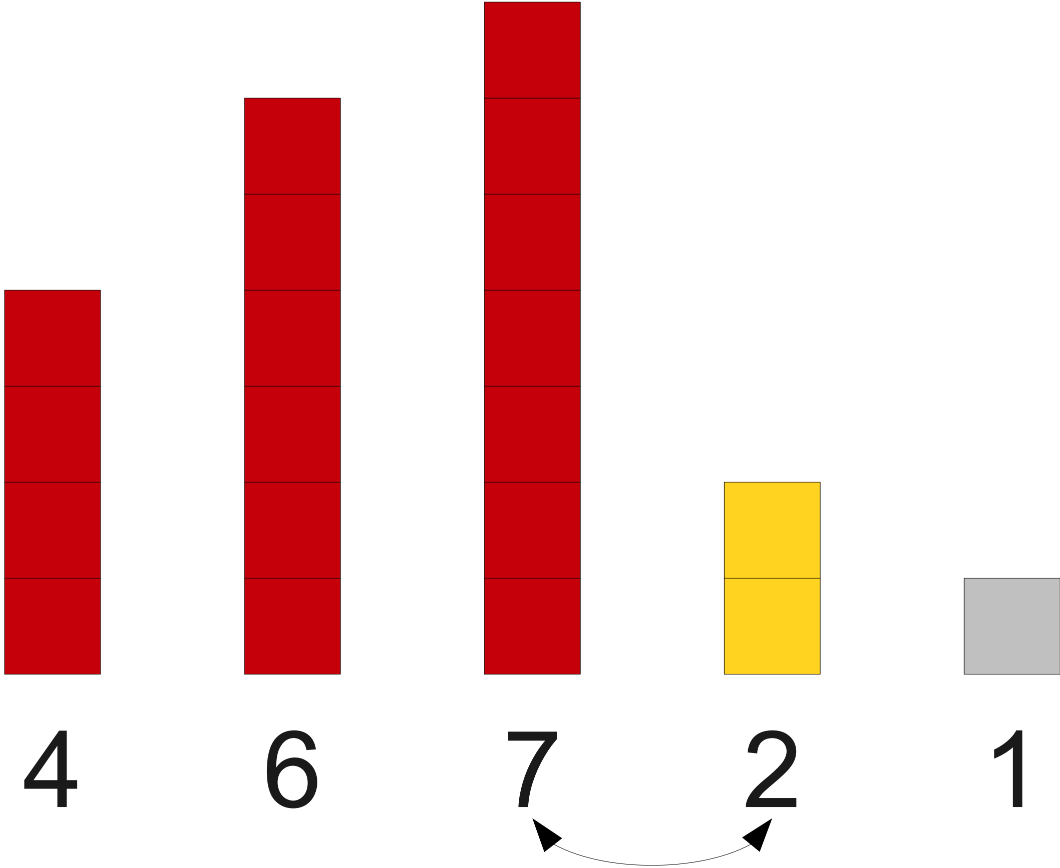
How Fast is Insertion Sort?



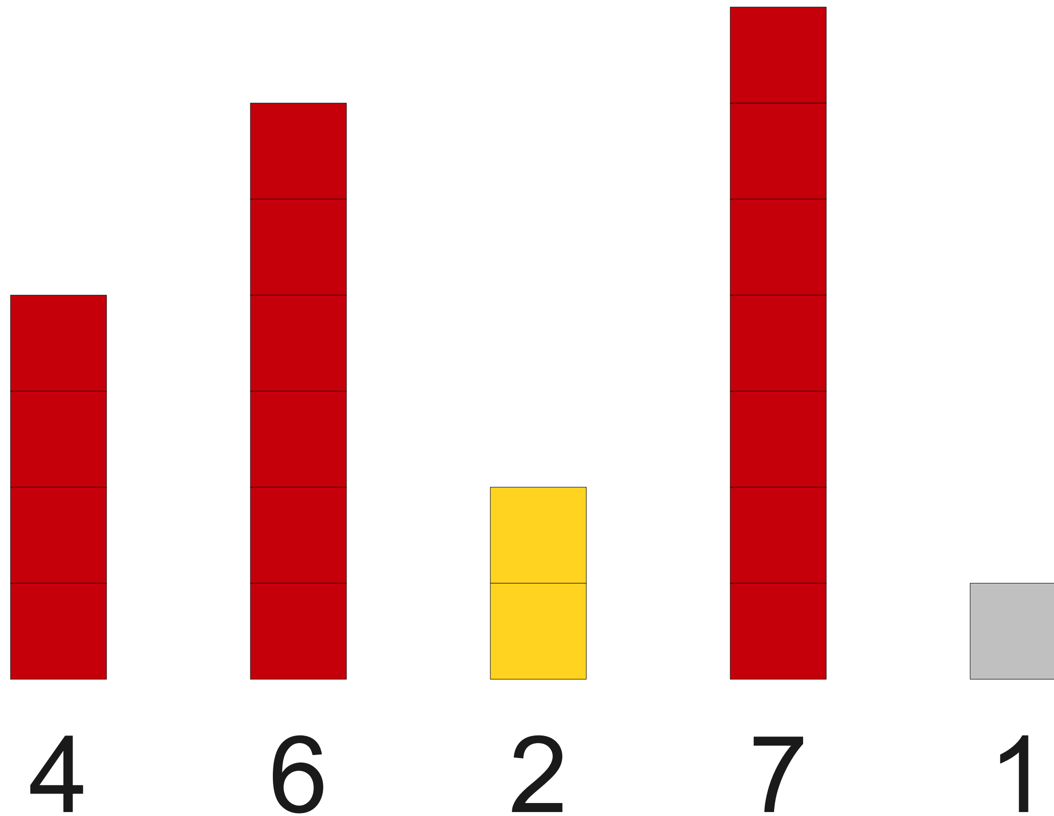
How Fast is Insertion Sort?



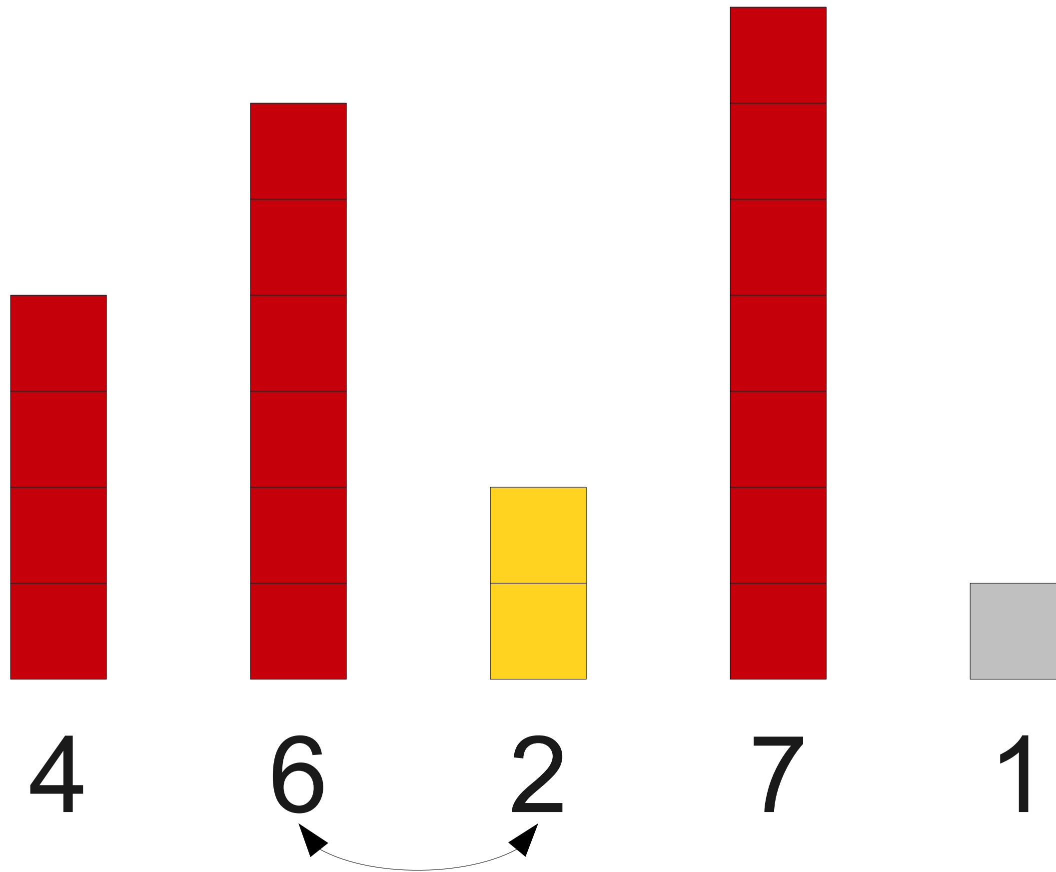
How Fast is Insertion Sort?



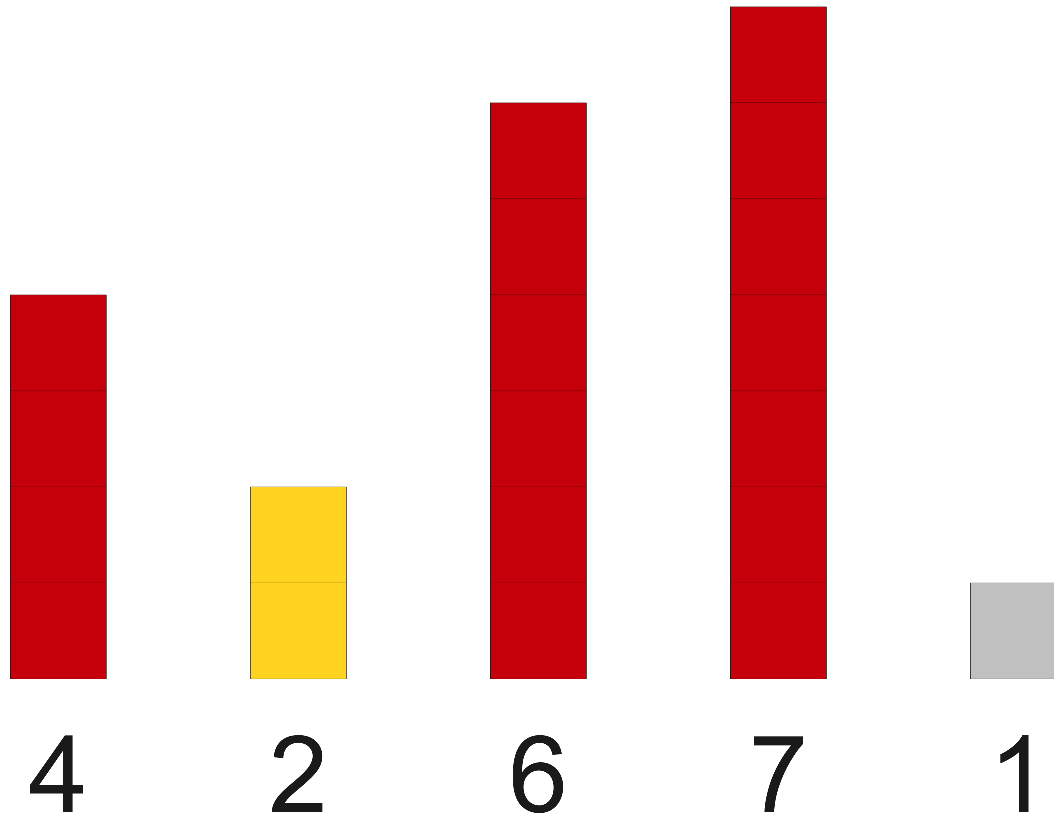
How Fast is Insertion Sort?



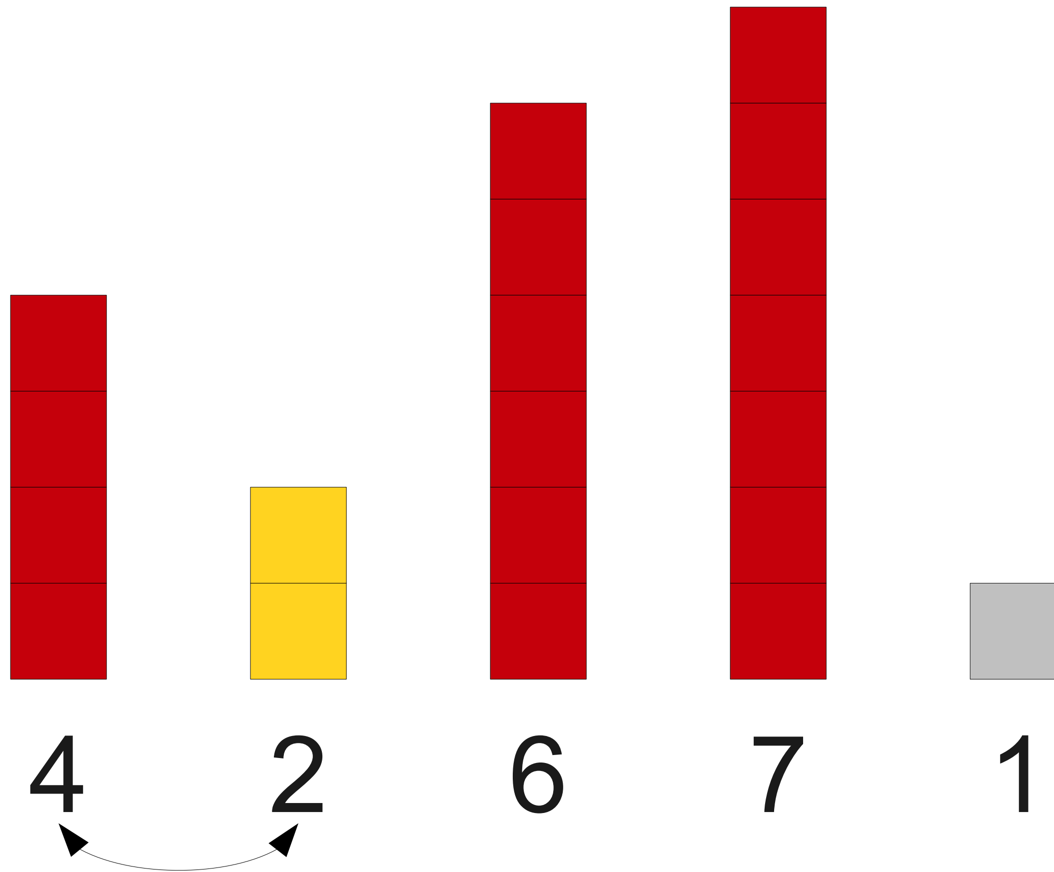
How Fast is Insertion Sort?



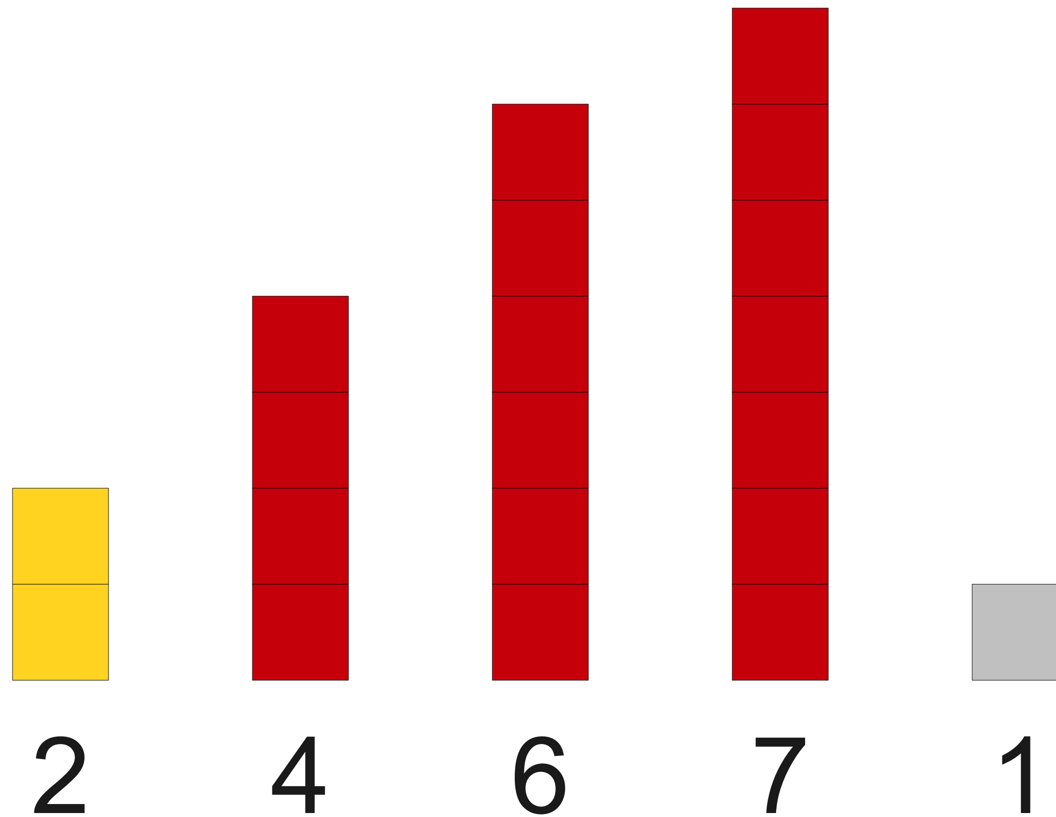
How Fast is Insertion Sort?



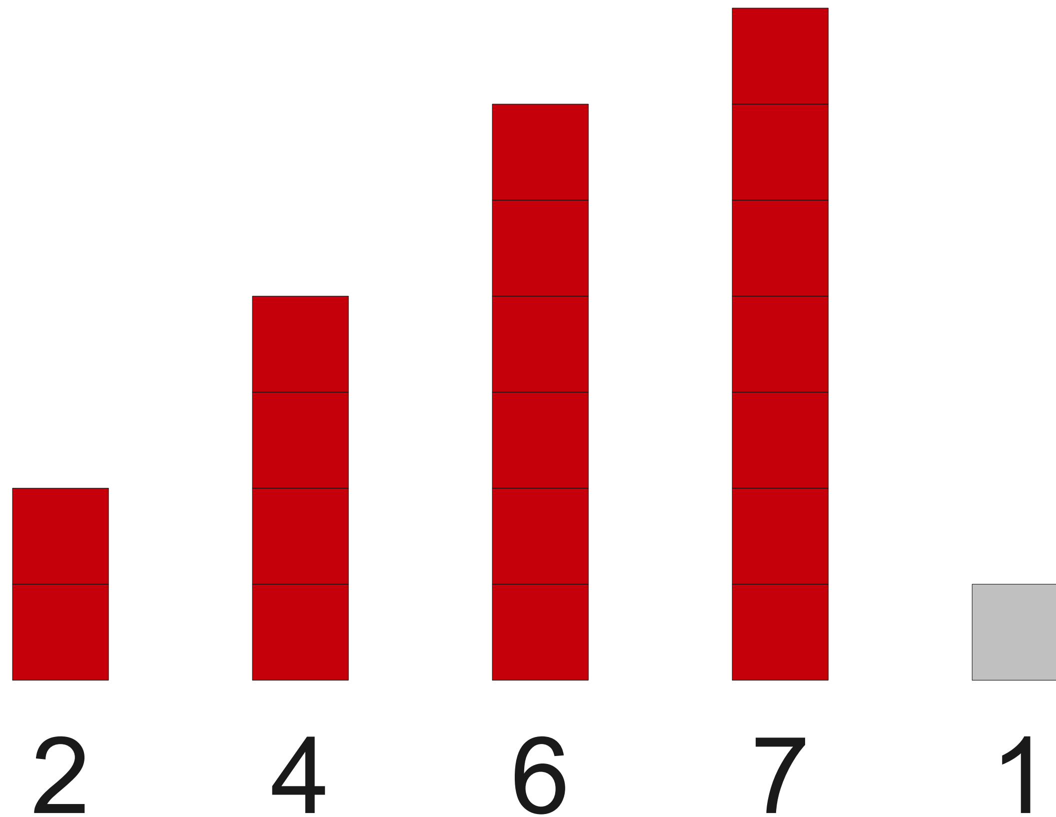
How Fast is Insertion Sort?



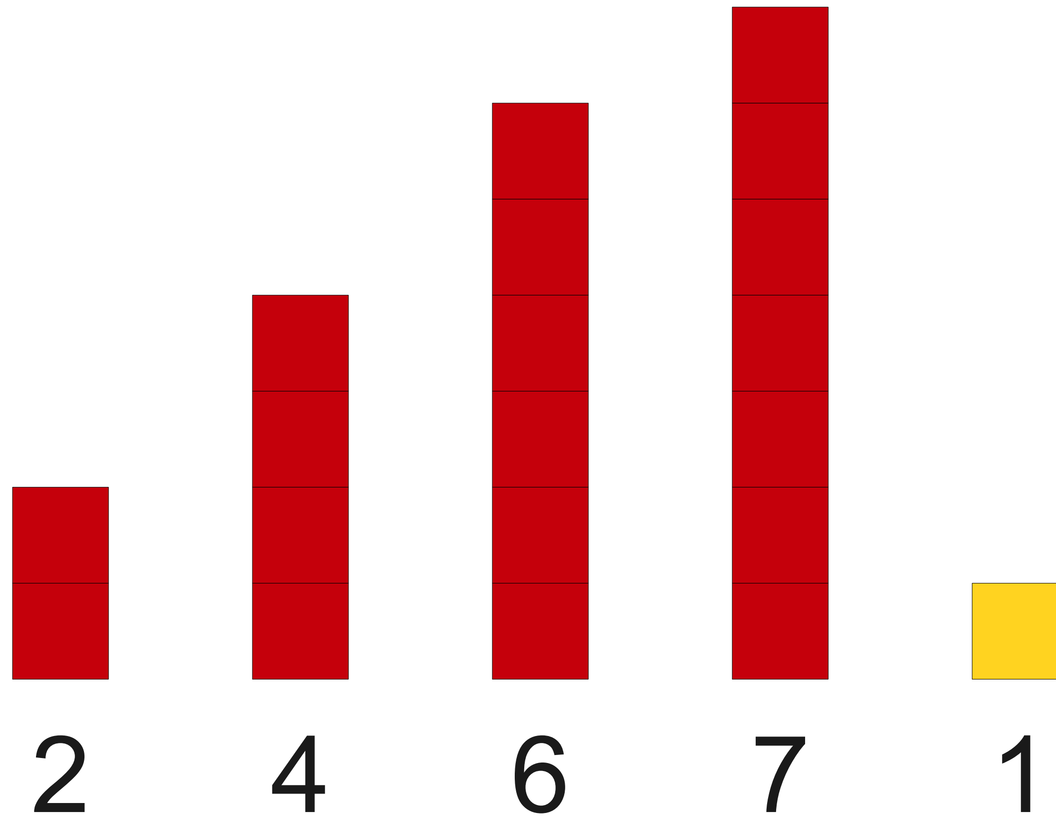
How Fast is Insertion Sort?



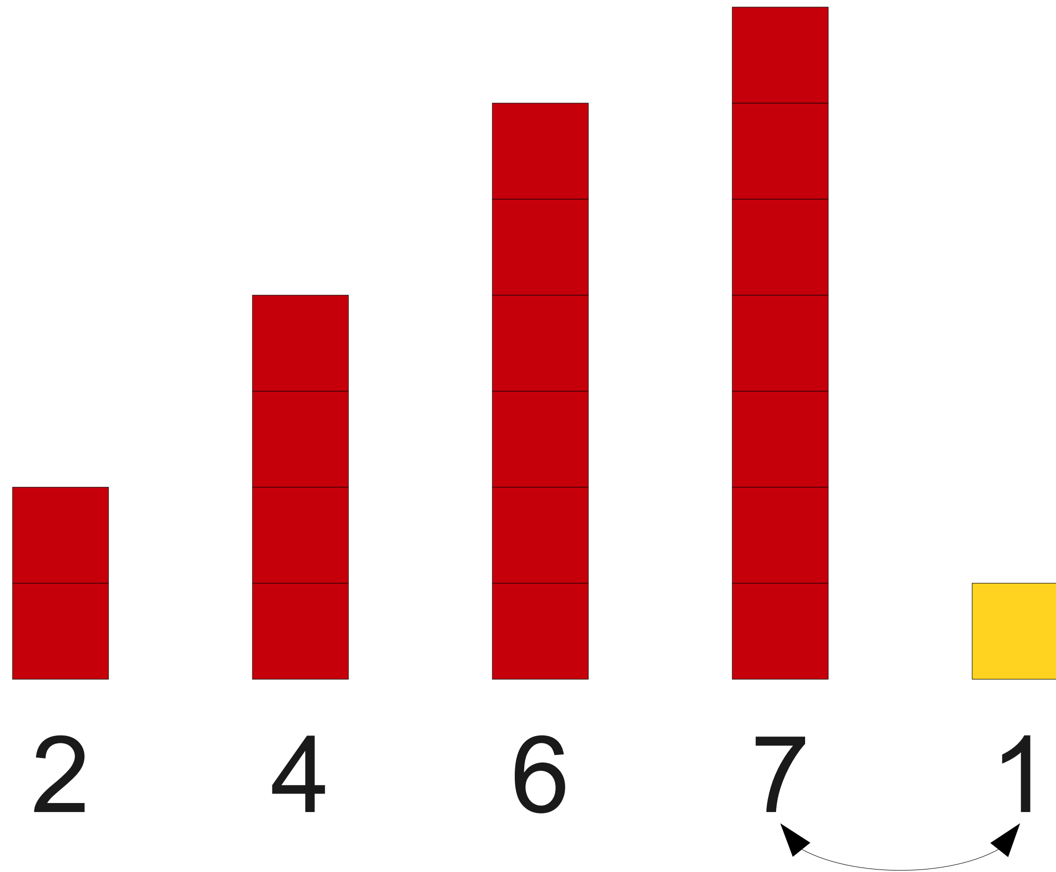
How Fast is Insertion Sort?



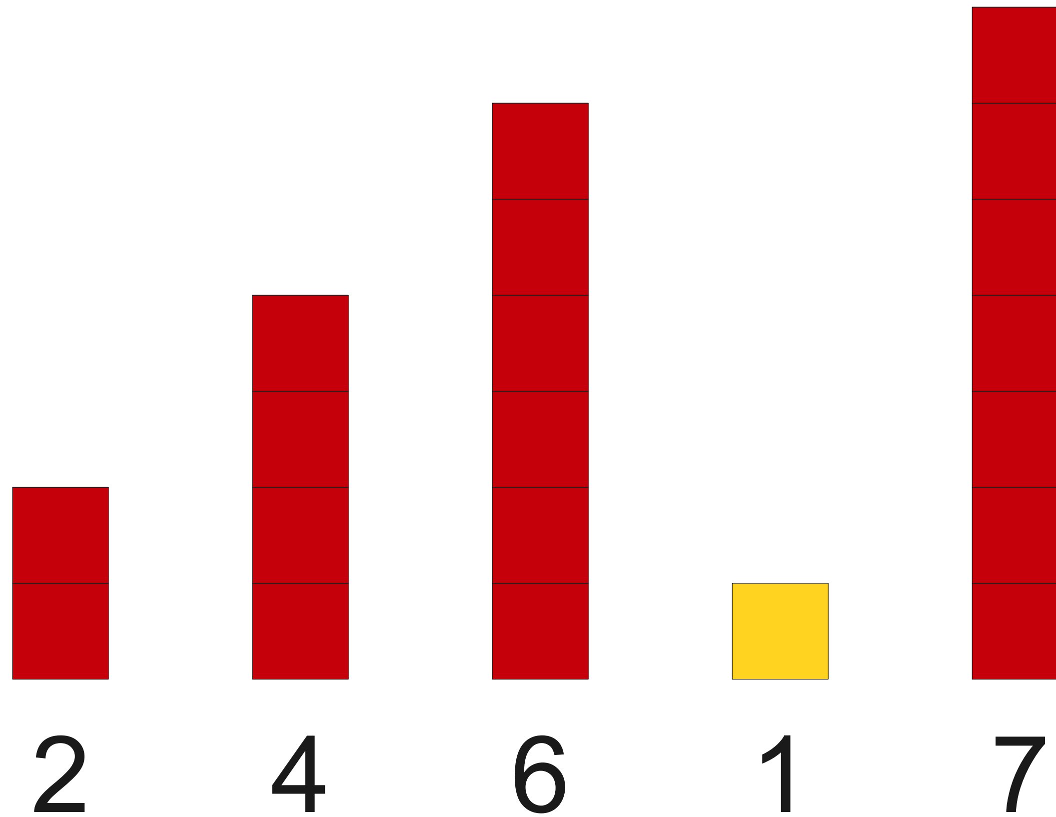
How Fast is Insertion Sort?



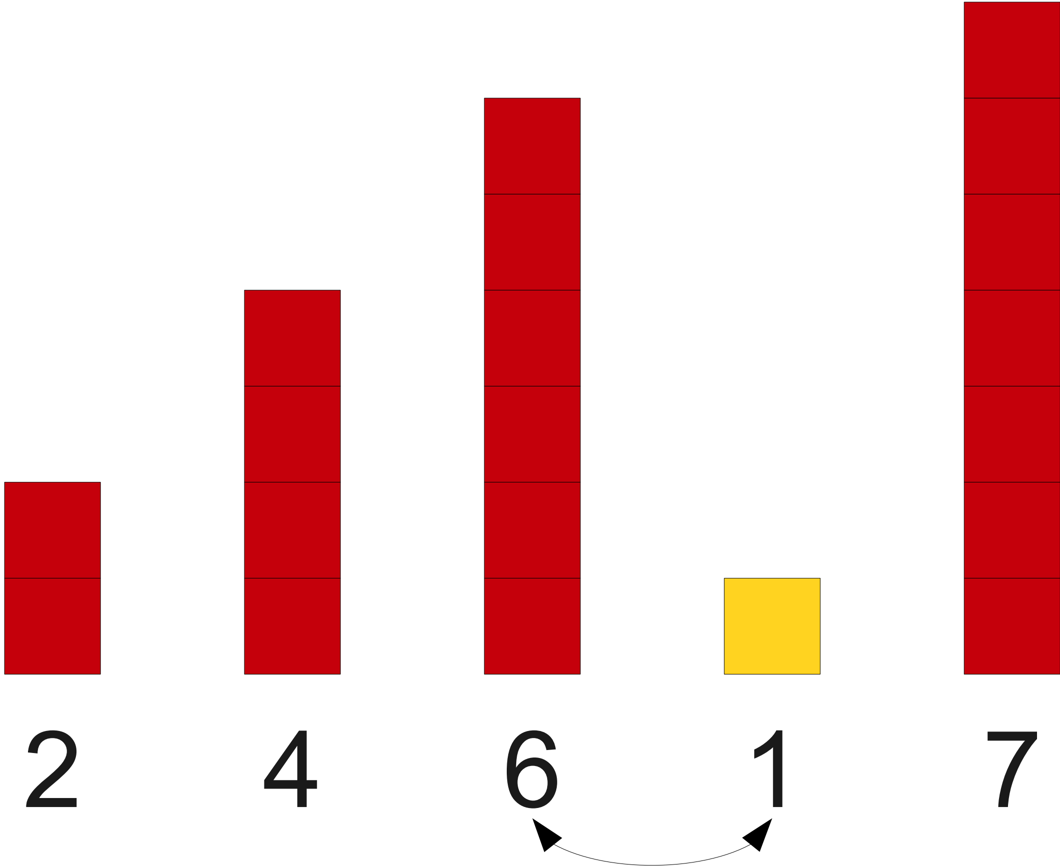
How Fast is Insertion Sort?



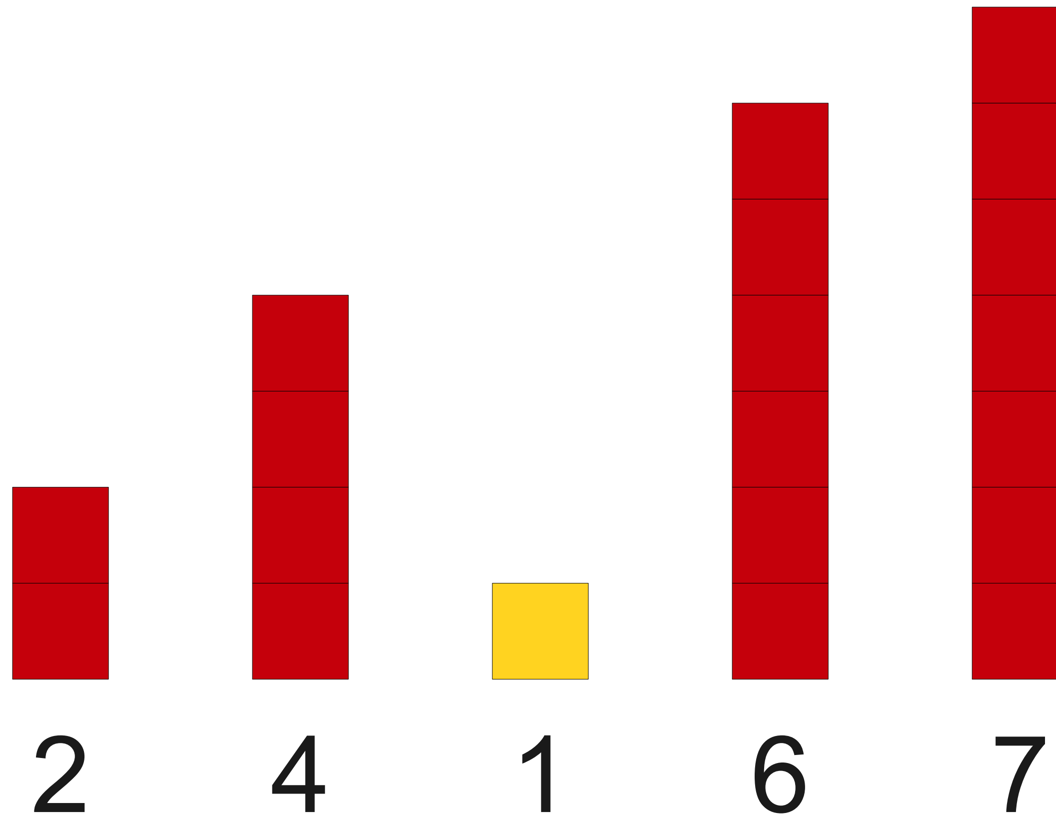
How Fast is Insertion Sort?



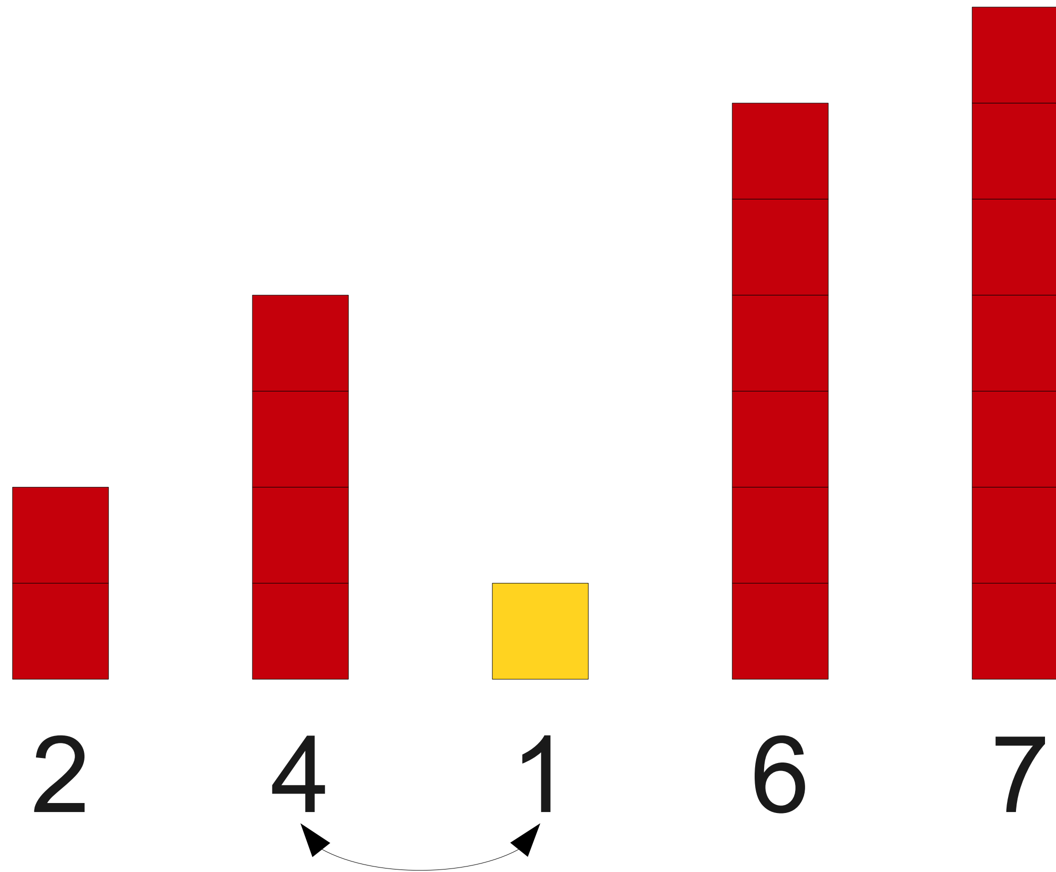
How Fast is Insertion Sort?



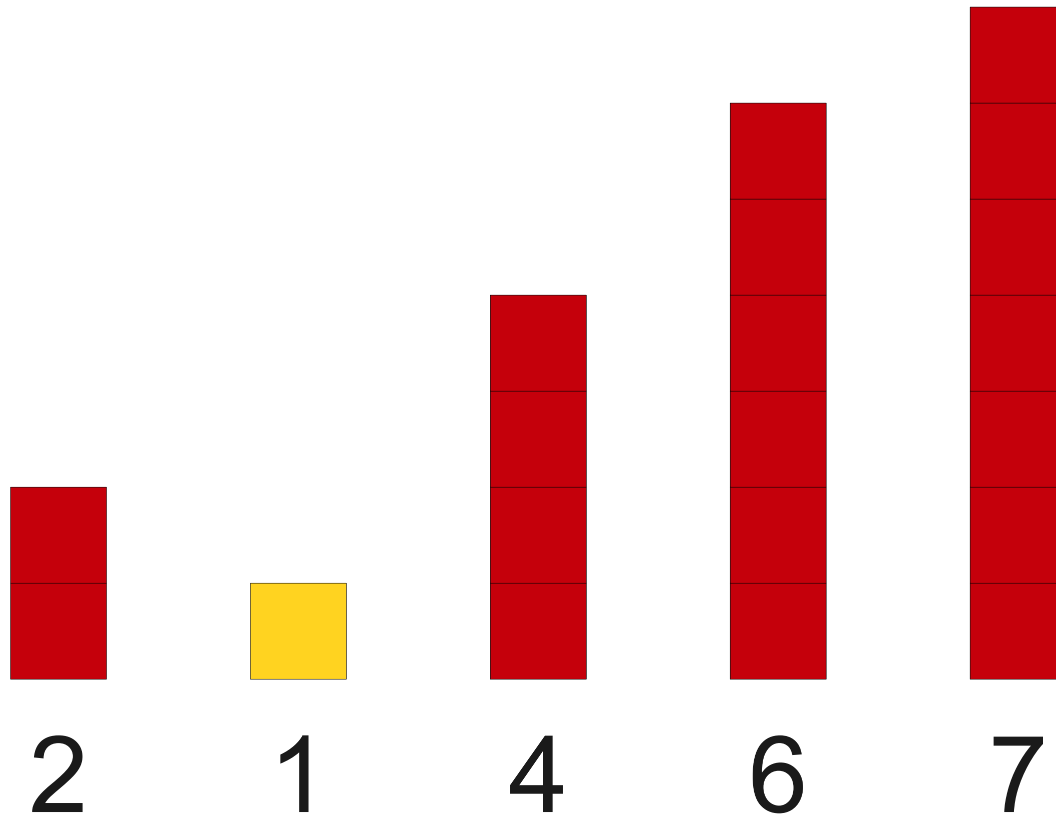
How Fast is Insertion Sort?



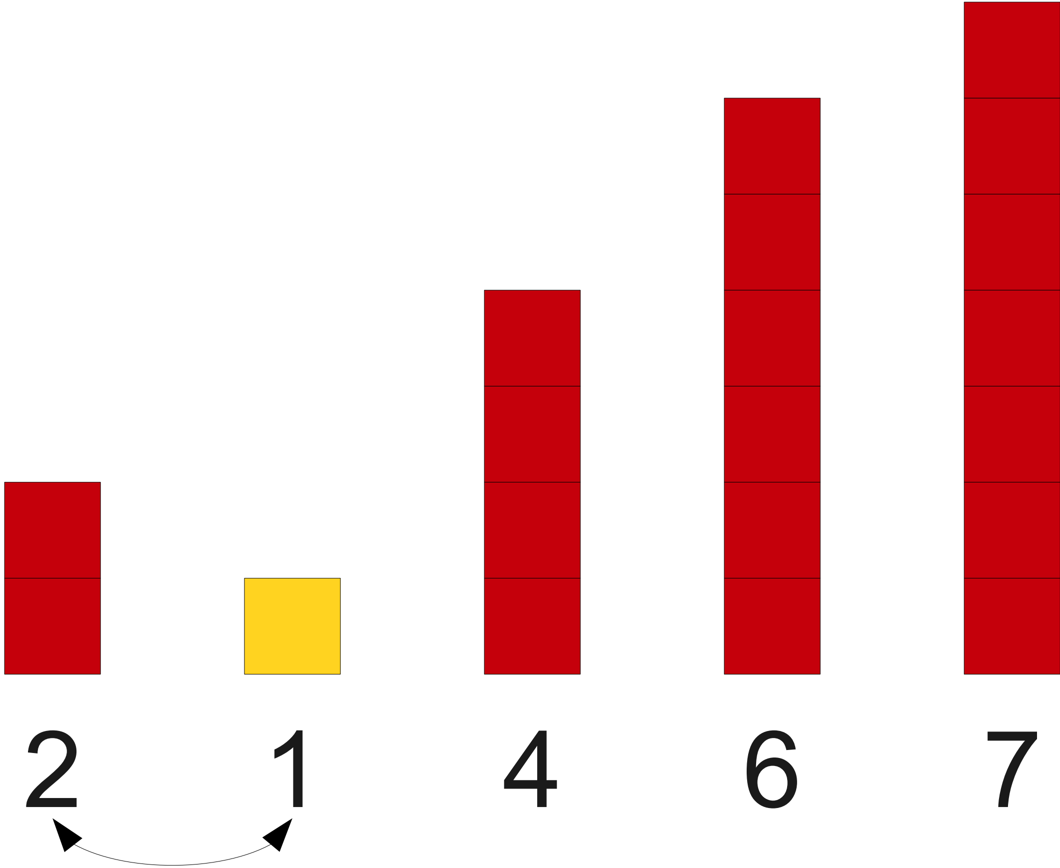
How Fast is Insertion Sort?



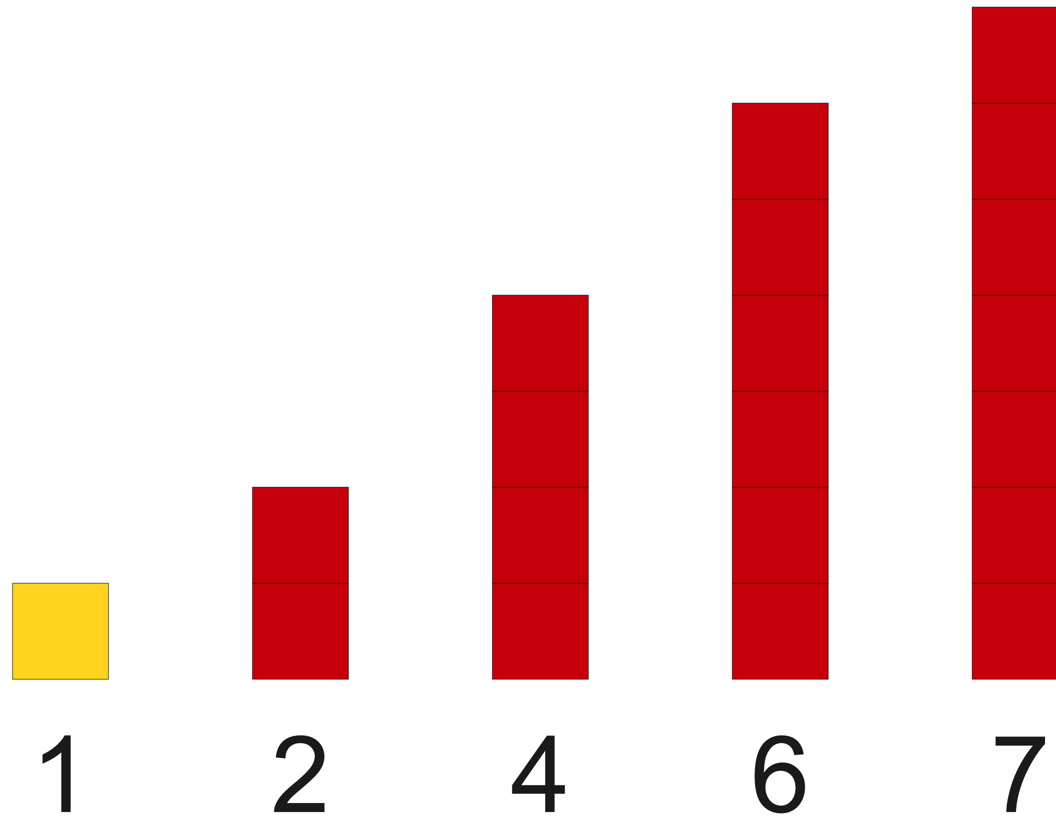
How Fast is Insertion Sort?



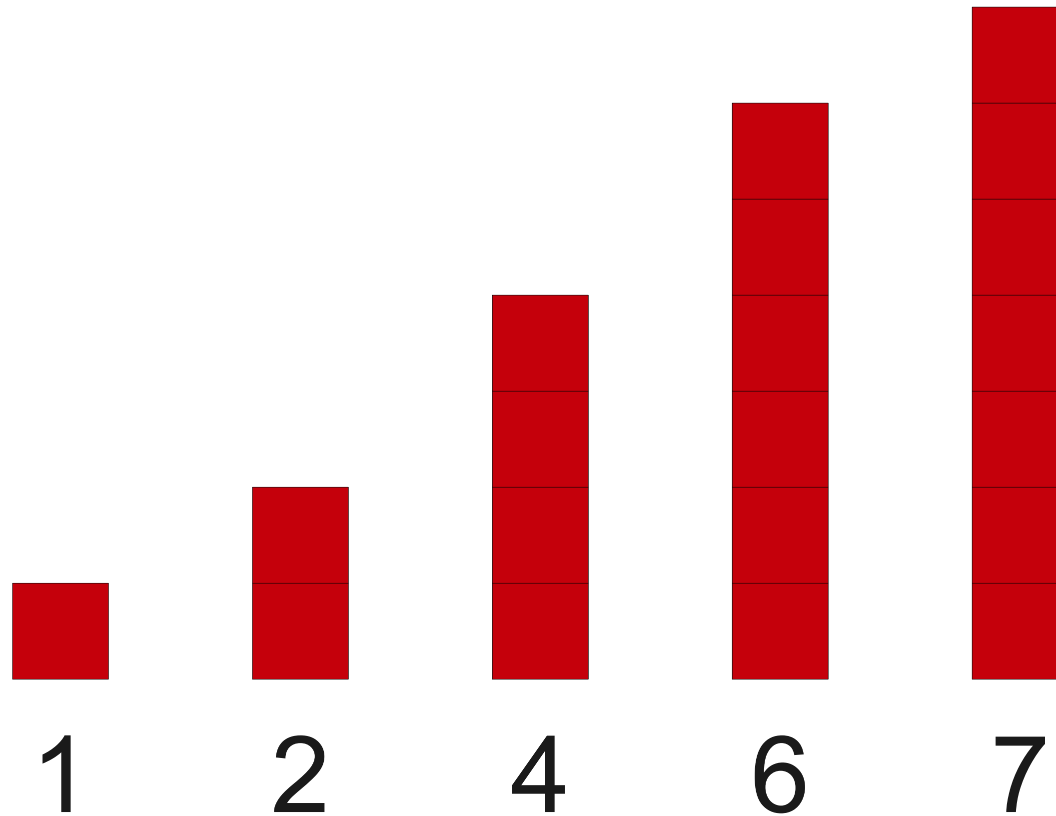
How Fast is Insertion Sort?



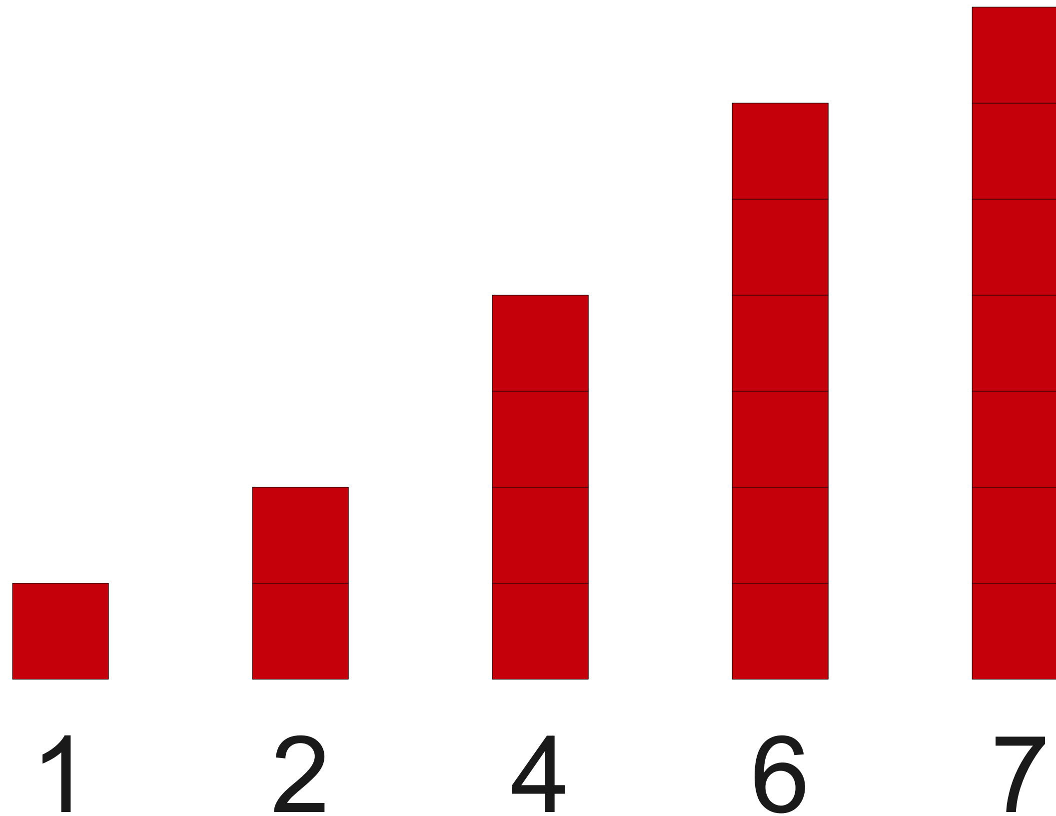
How Fast is Insertion Sort?



How Fast is Insertion Sort?



How Fast is Insertion Sort?



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

Notes on Insertion Sort

- Insertion sort is O(n) in the best case.
- Insertion sort is $O(n^2)$ 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

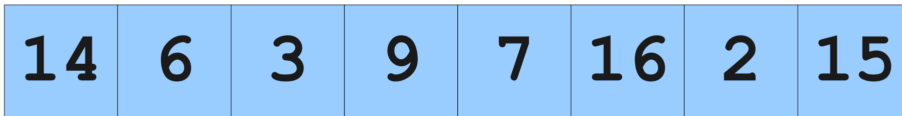
Thinking About $O(n^2)$

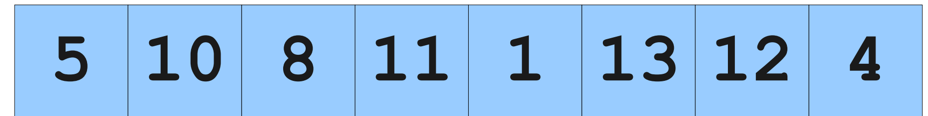
14	6	3	9	7	16	2	15	5	10	8	11	1	13	12	4

$T(n)$

Thinking About $O(n^2)$

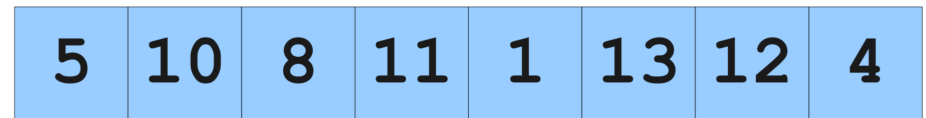
T(n)





Thinking About $O(n^2)$

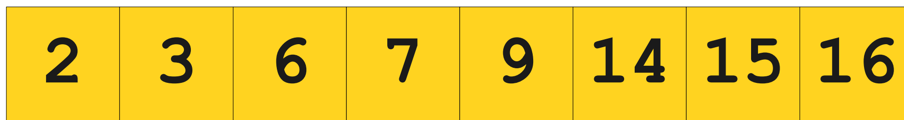
$T(n)$



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

Thinking About $O(n^2)$

$T(n)$

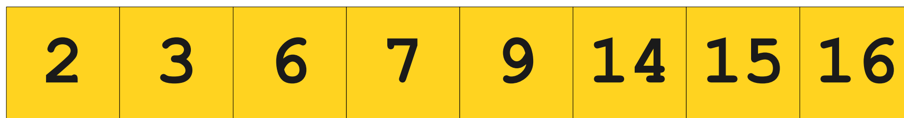


$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$

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

Thinking About $O(n^2)$

$T(n)$



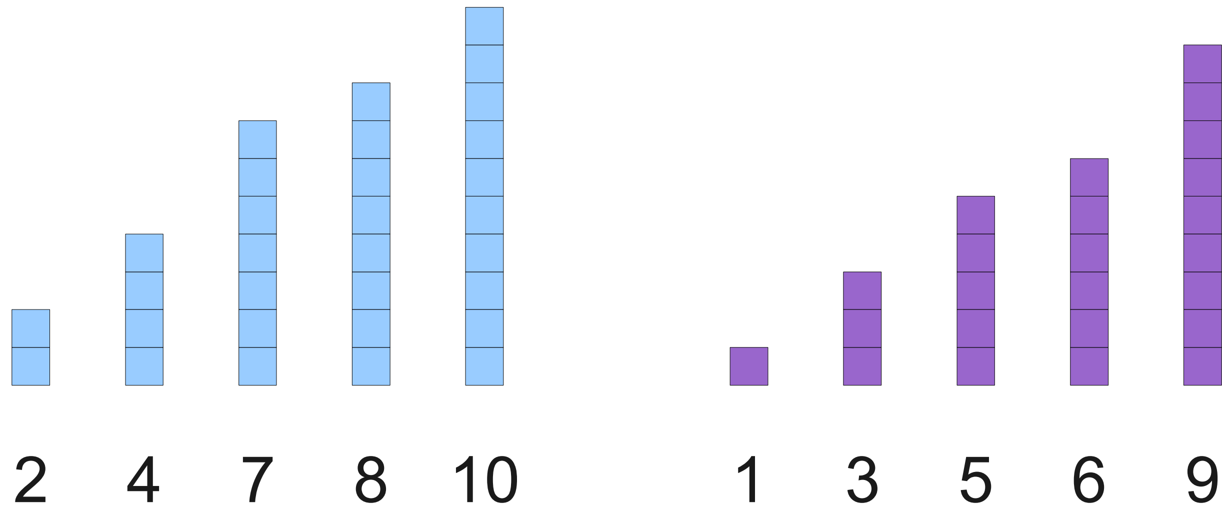
$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$

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

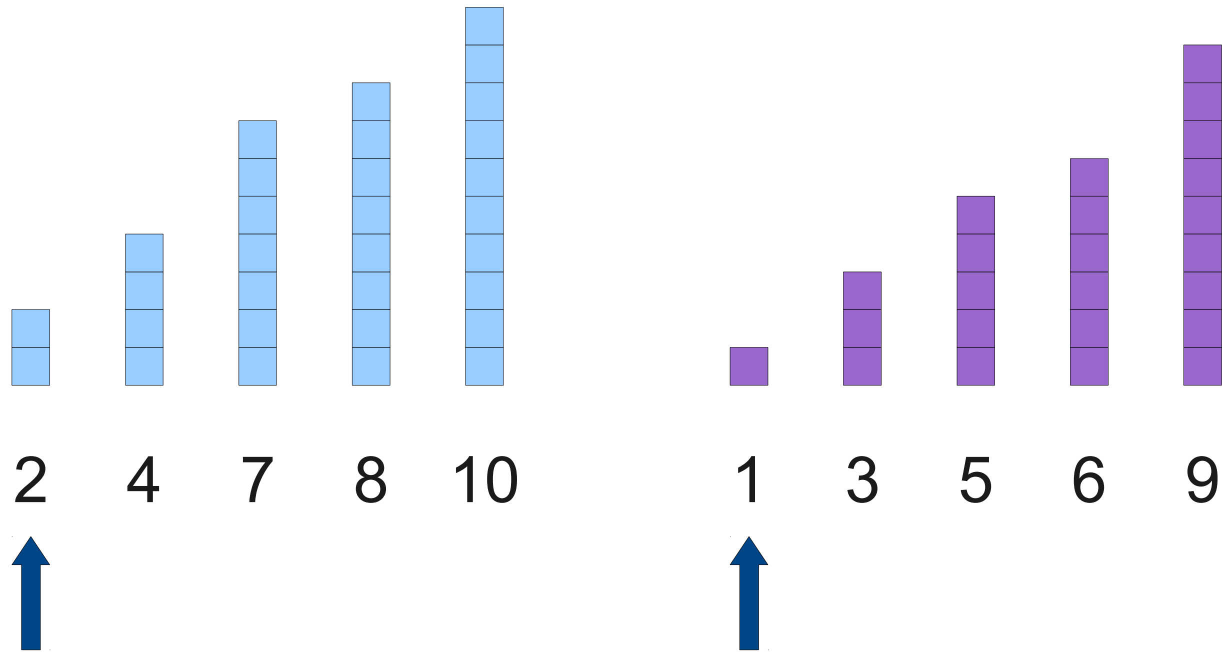
It takes roughly ½T(n) to sort each half separately!

The Key Insight: **Merge**

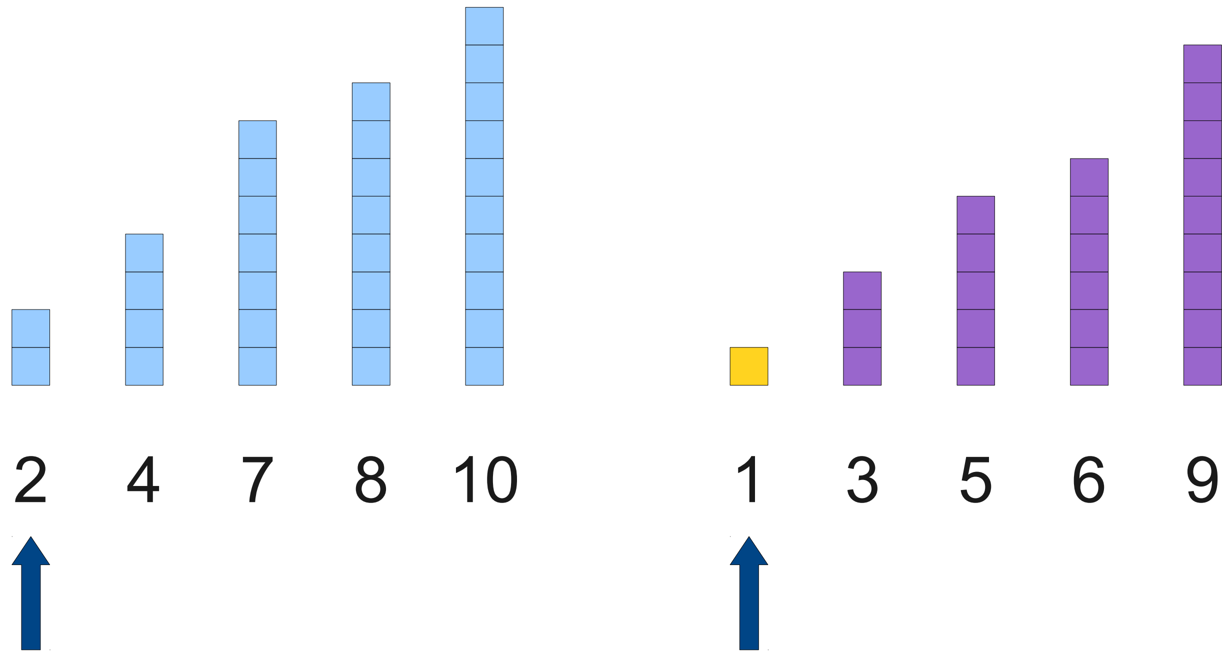
The Key Insight: **Merge**



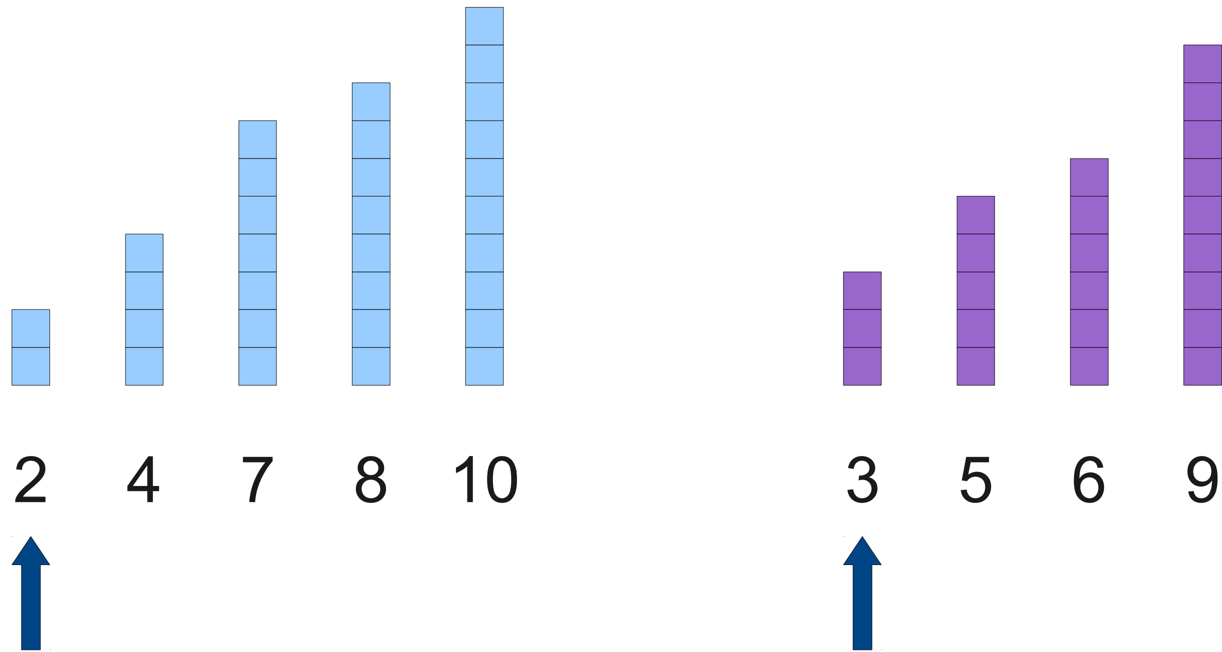
The Key Insight: **Merge**



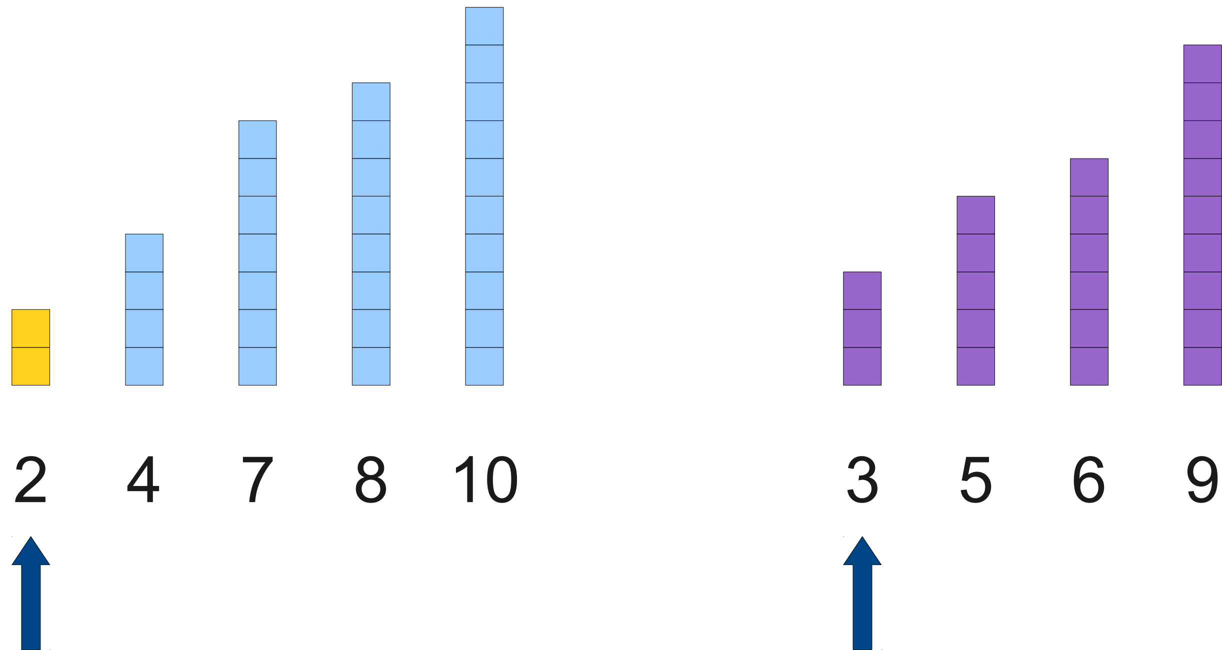
The Key Insight: **Merge**



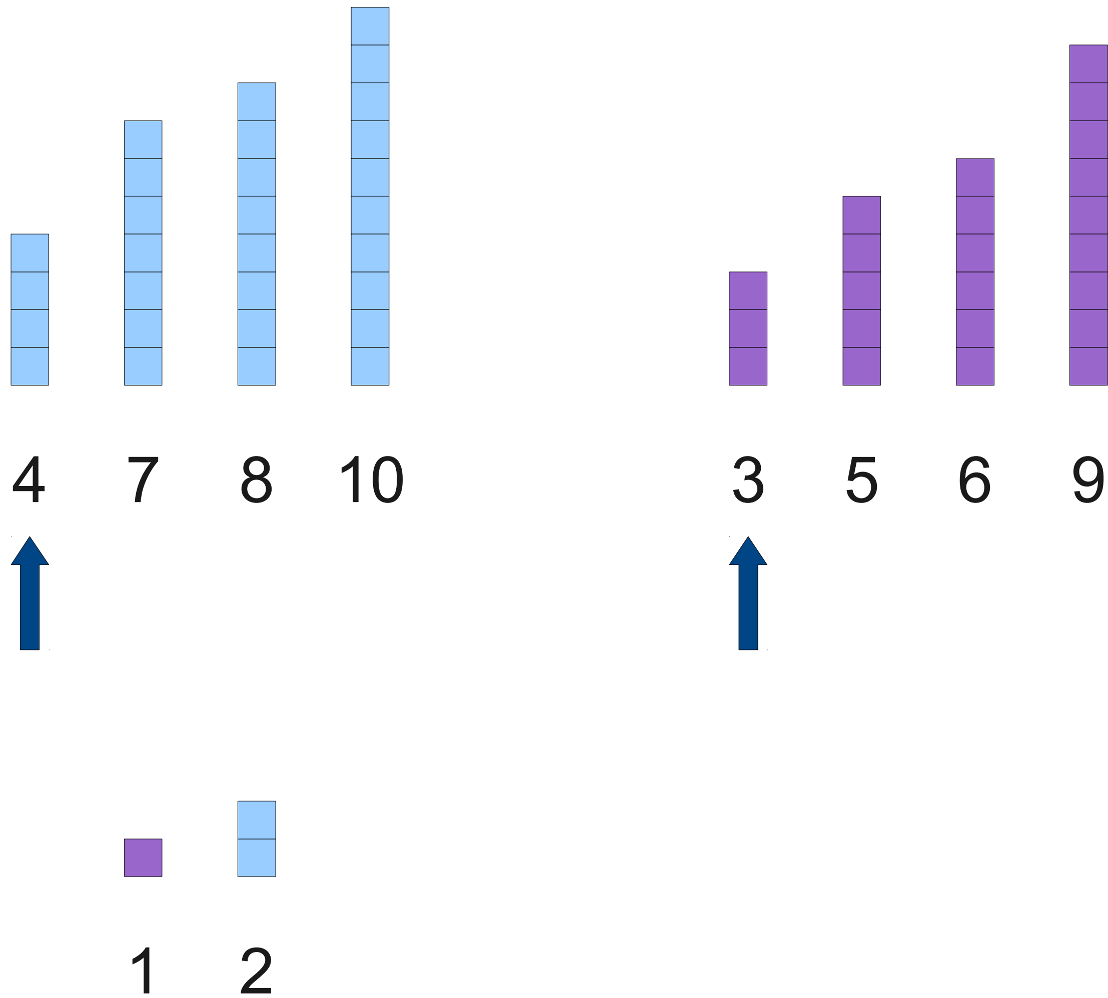
The Key Insight: **Merge**



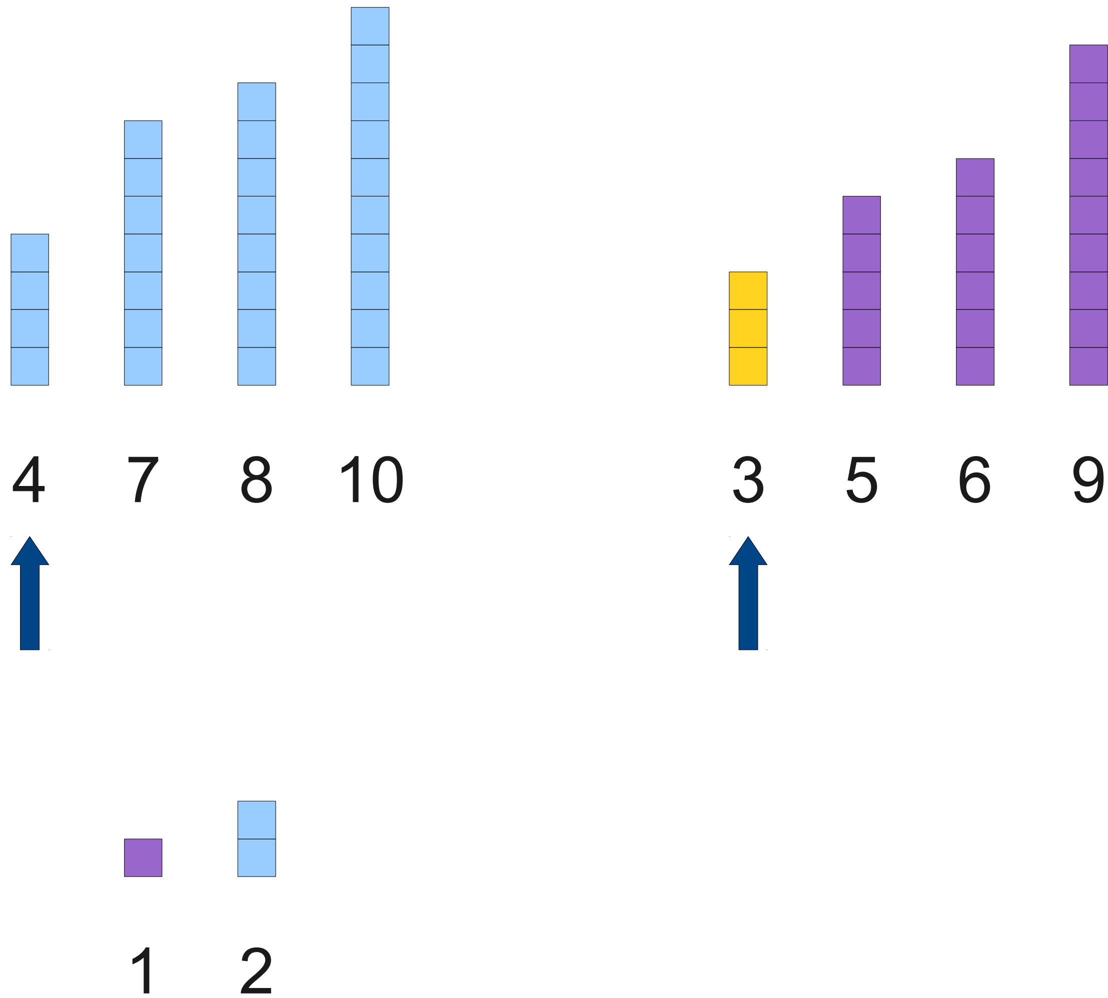
The Key Insight: Merge



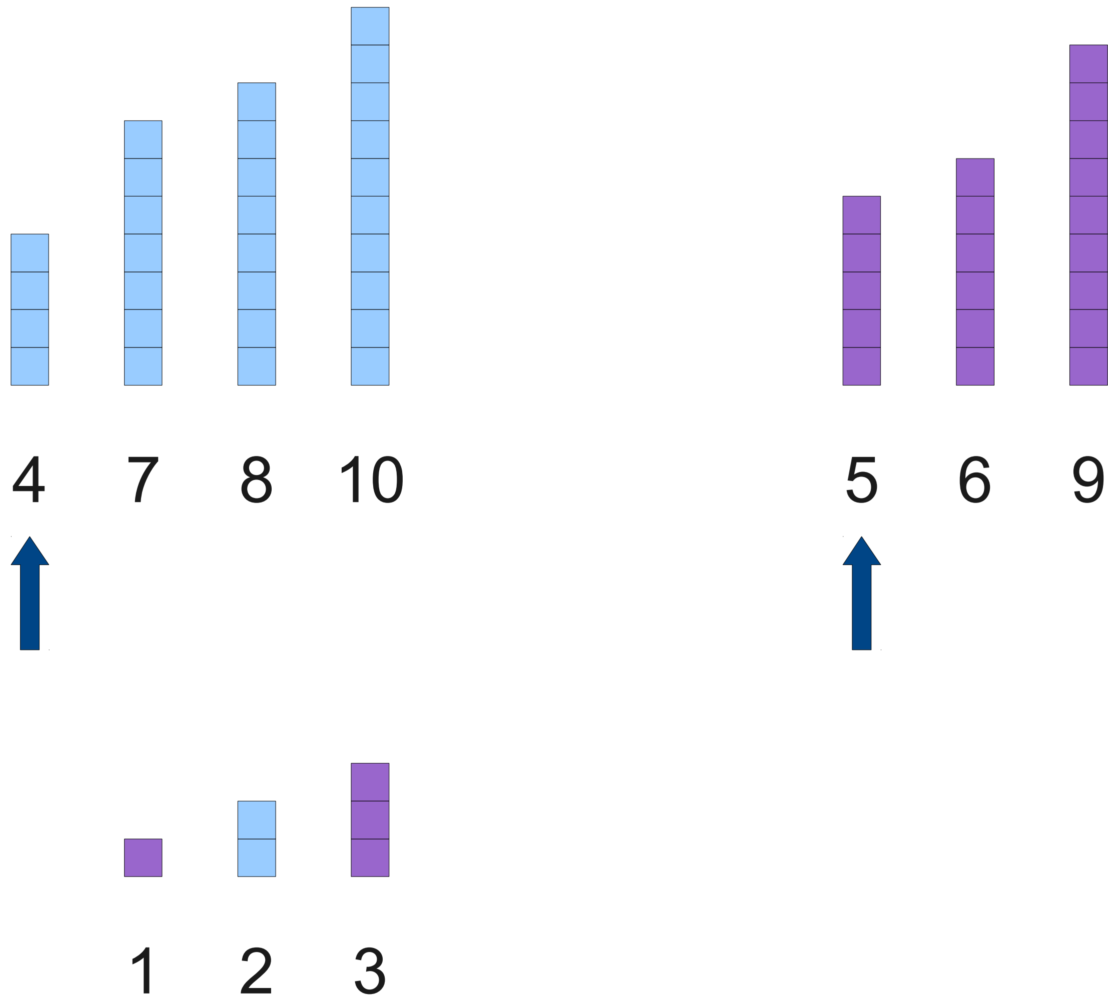
The Key Insight: **Merge**



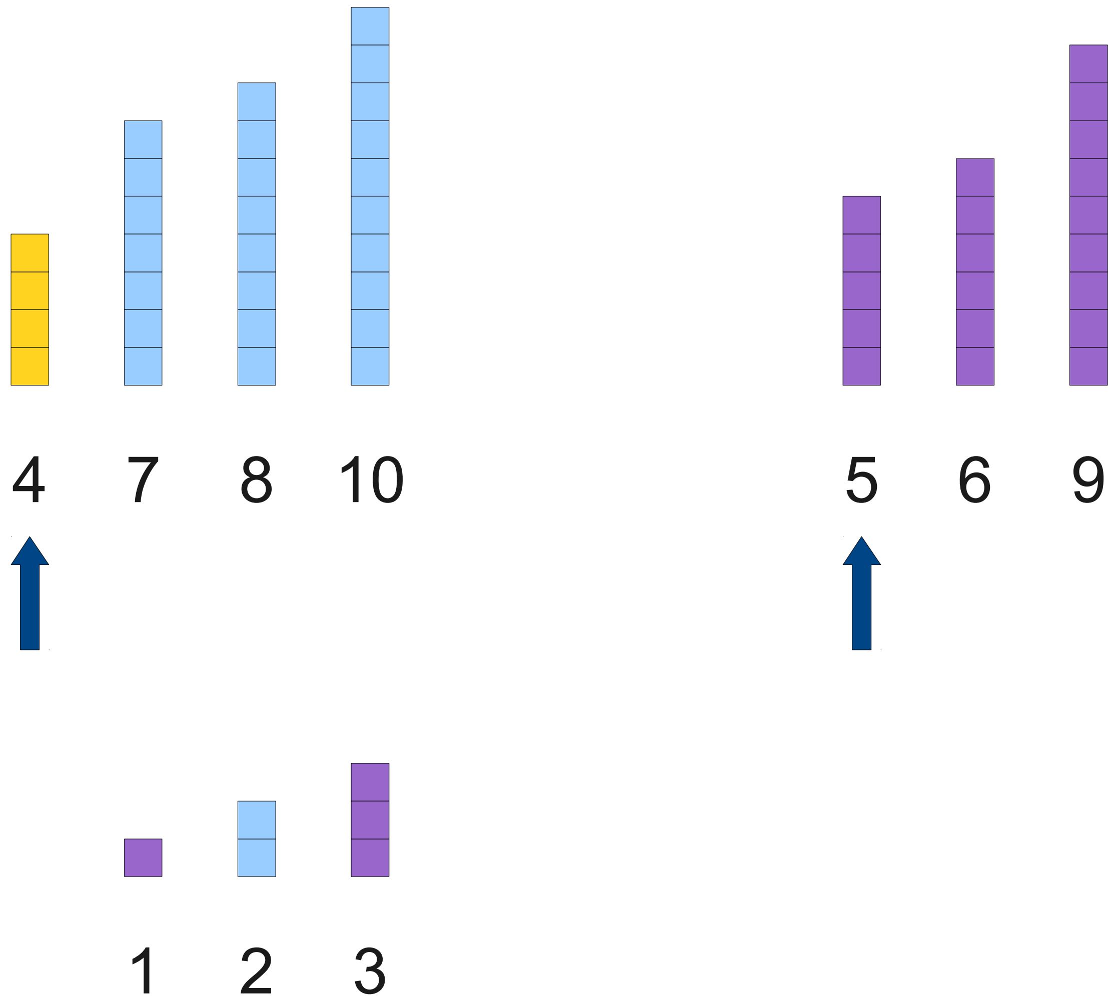
The Key Insight: **Merge**



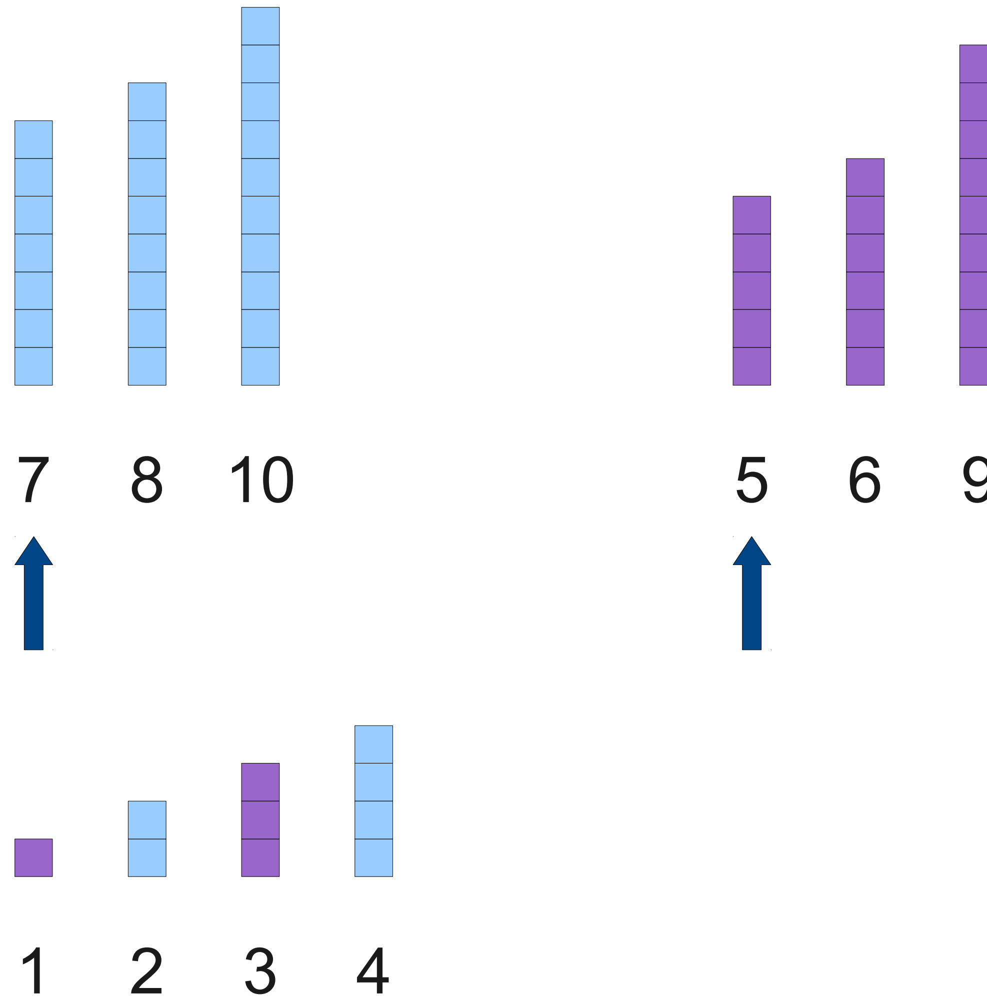
The Key Insight: **Merge**



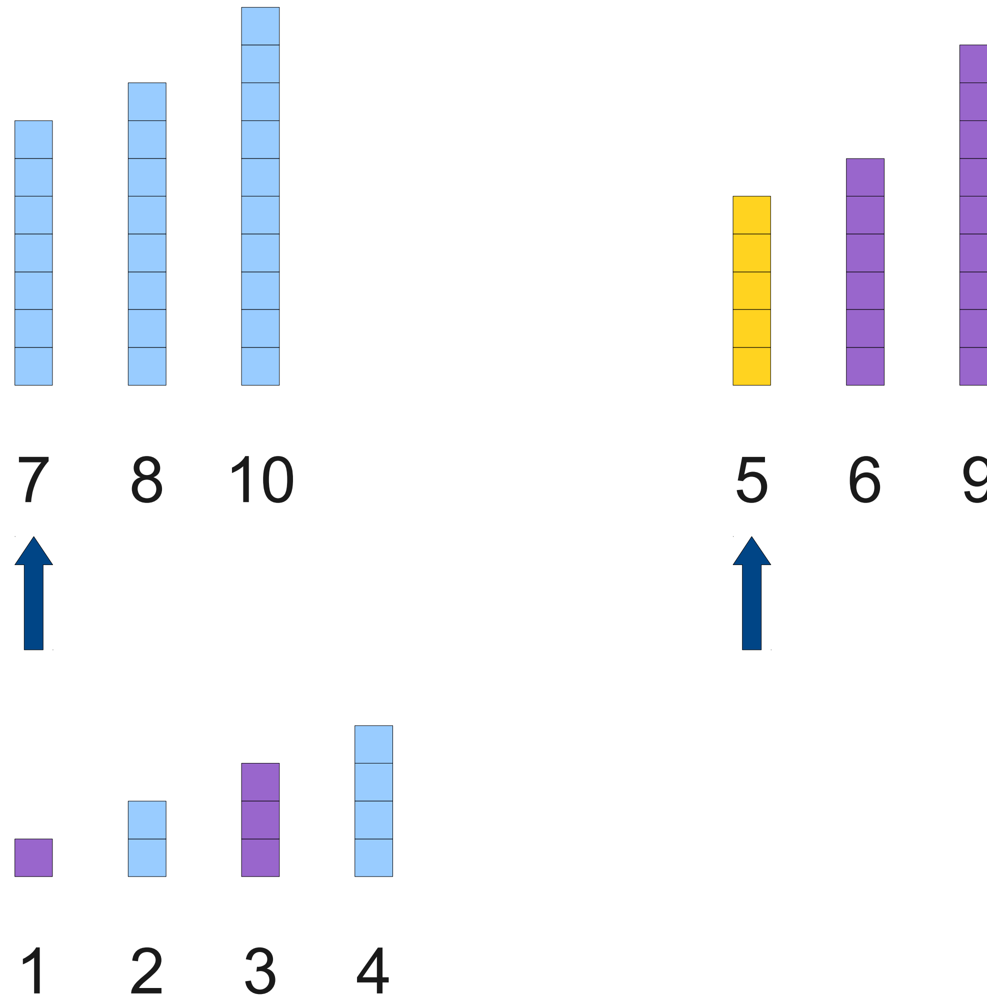
The Key Insight: **Merge**



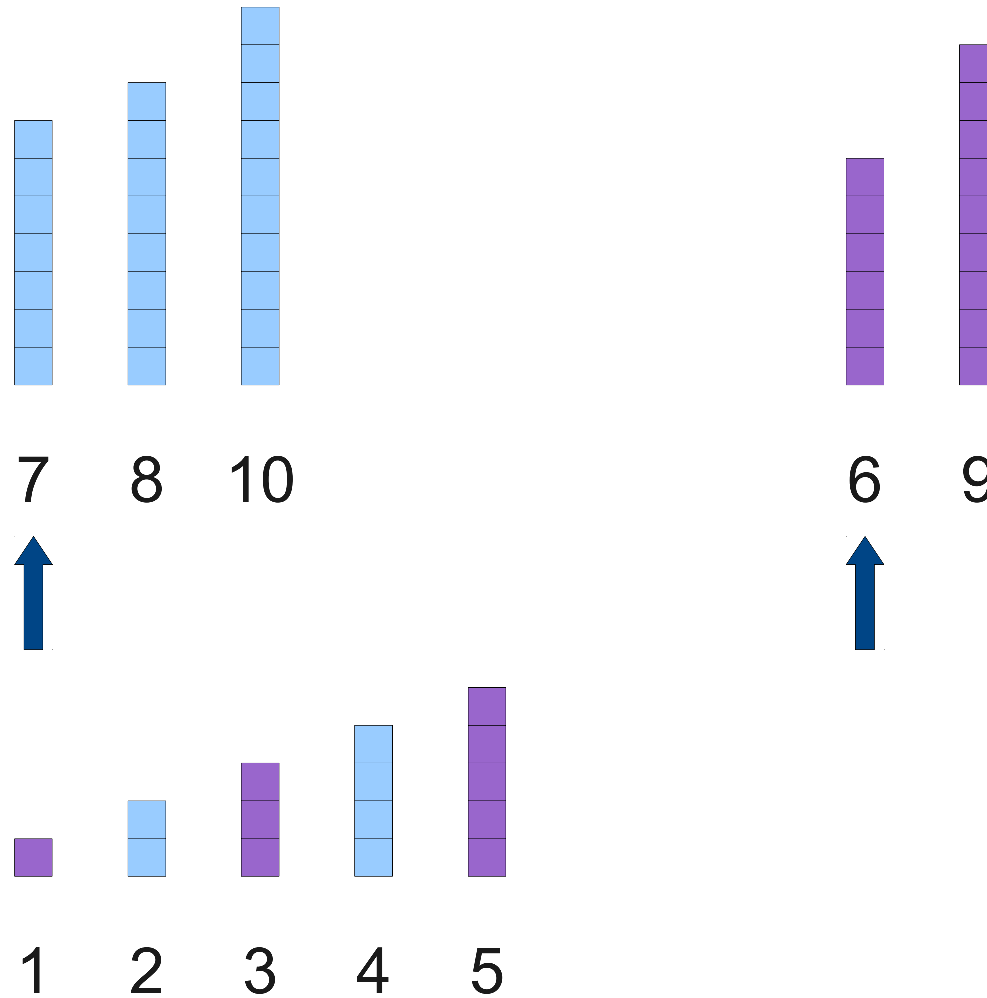
The Key Insight: **Merge**



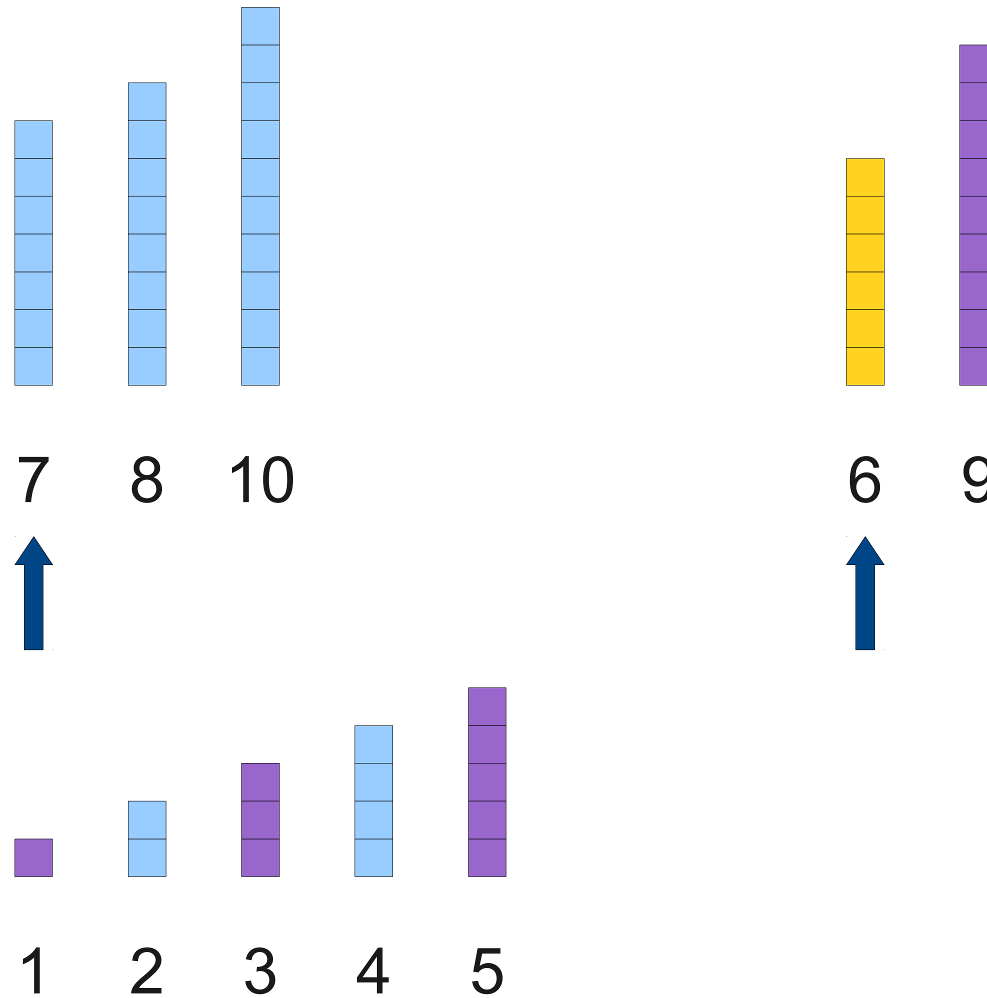
The Key Insight: **Merge**



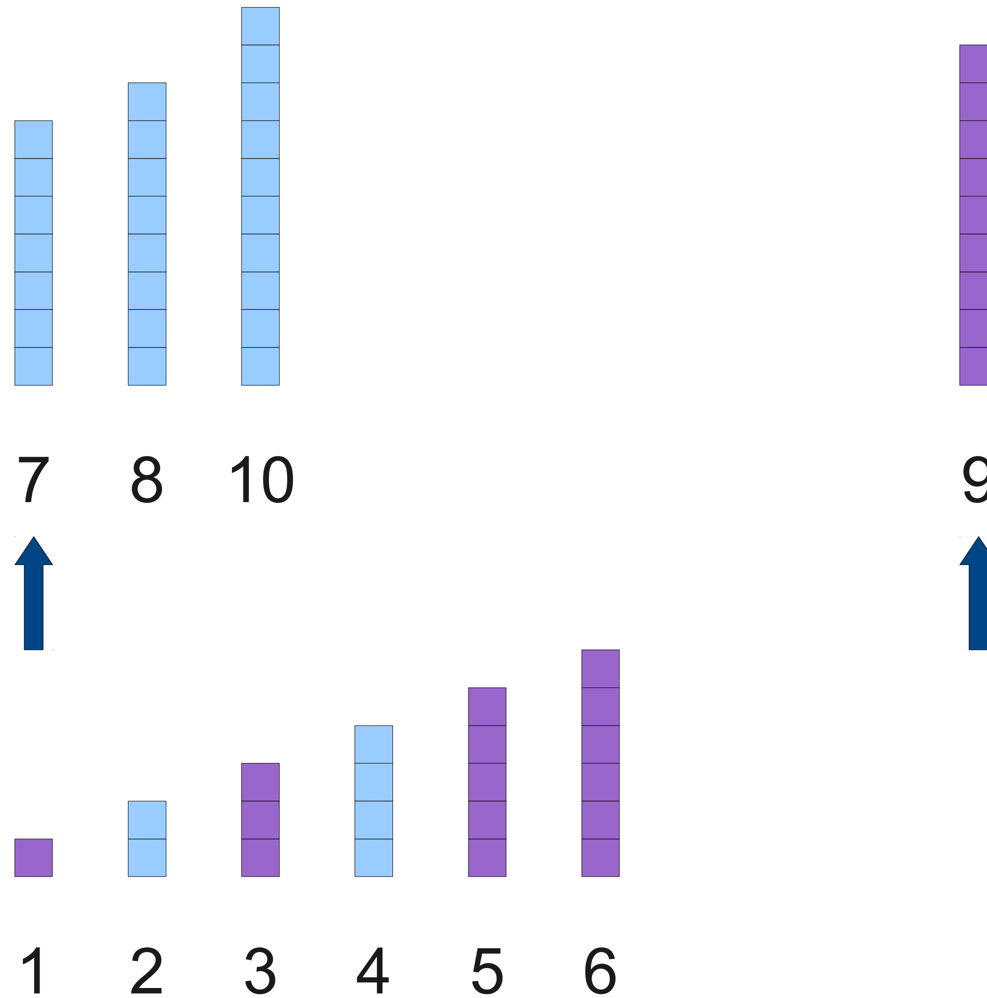
The Key Insight: **Merge**



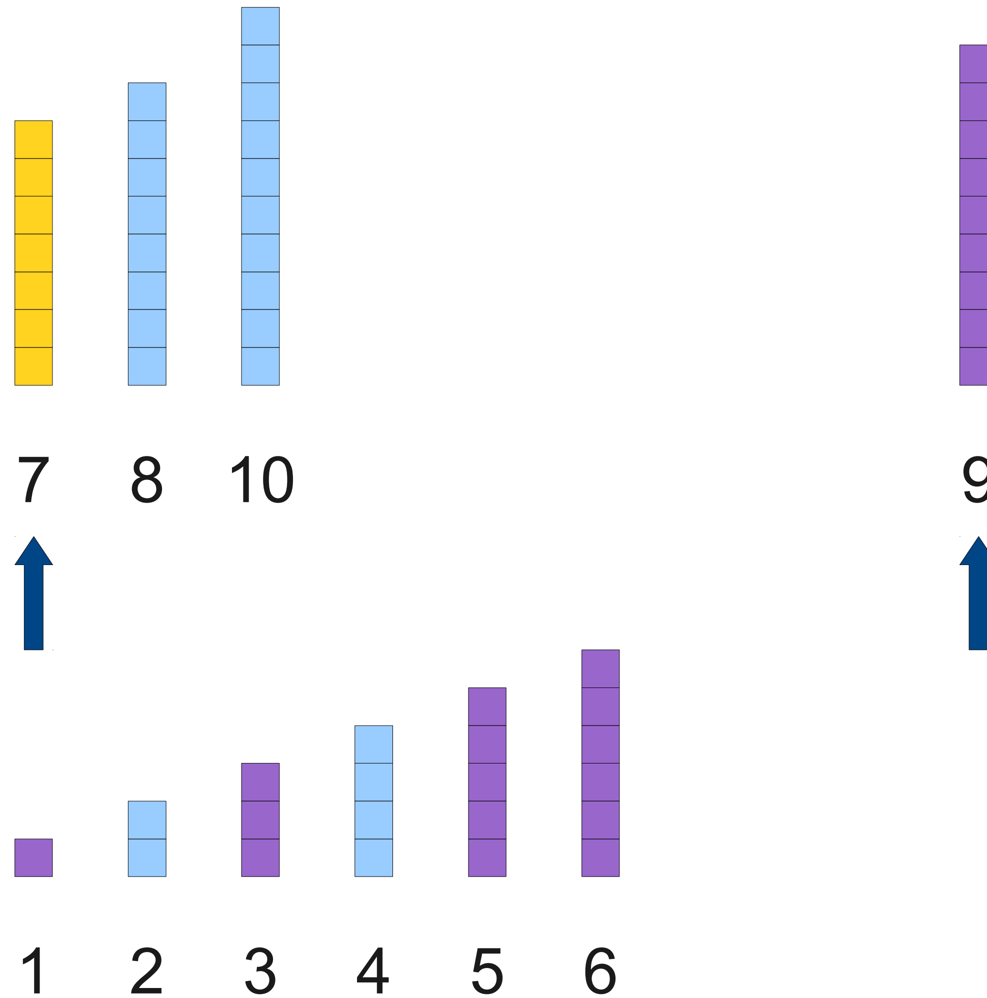
The Key Insight: **Merge**



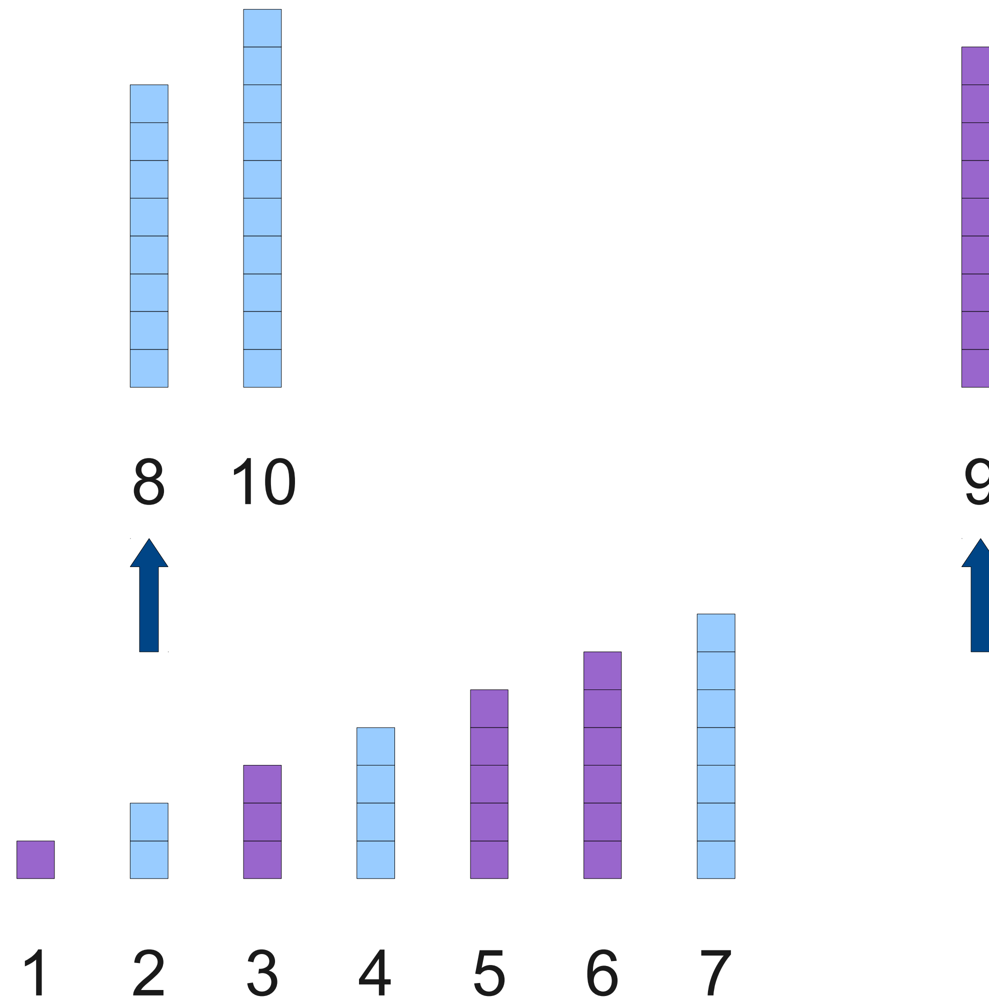
The Key Insight: **Merge**



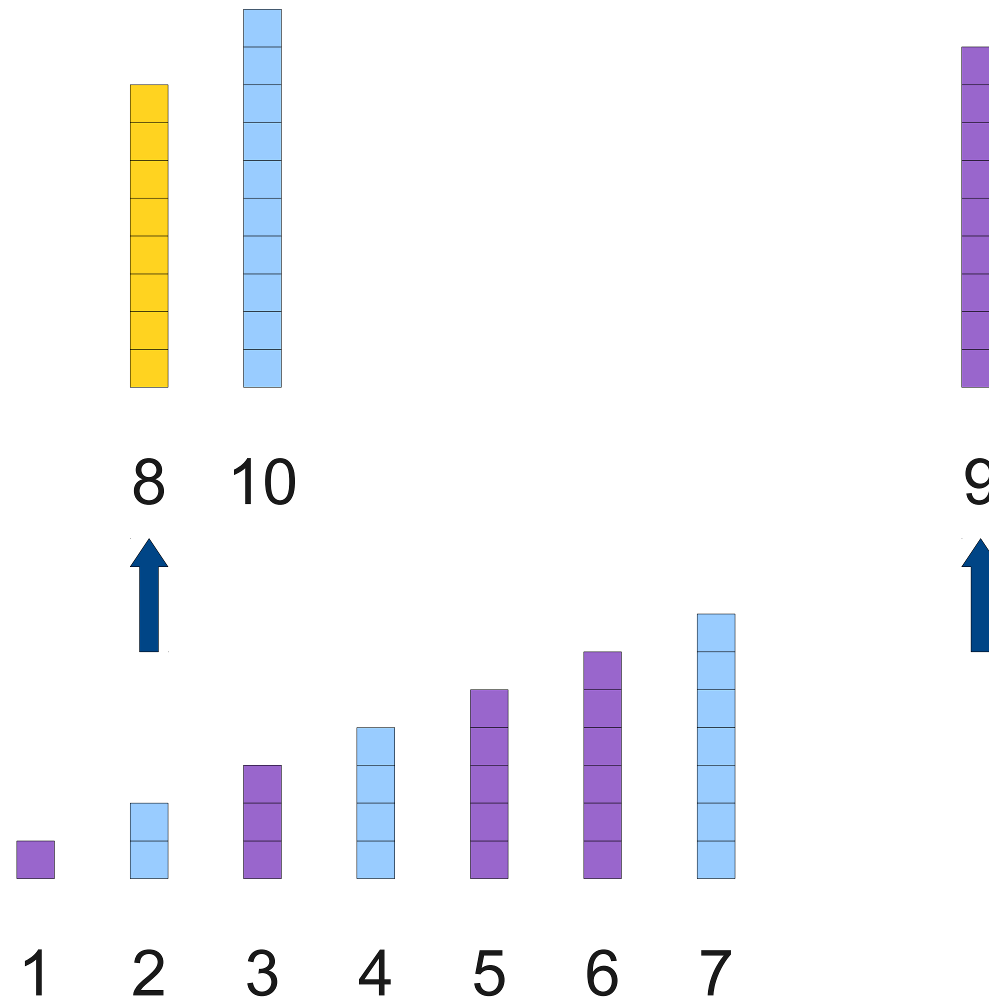
The Key Insight: **Merge**



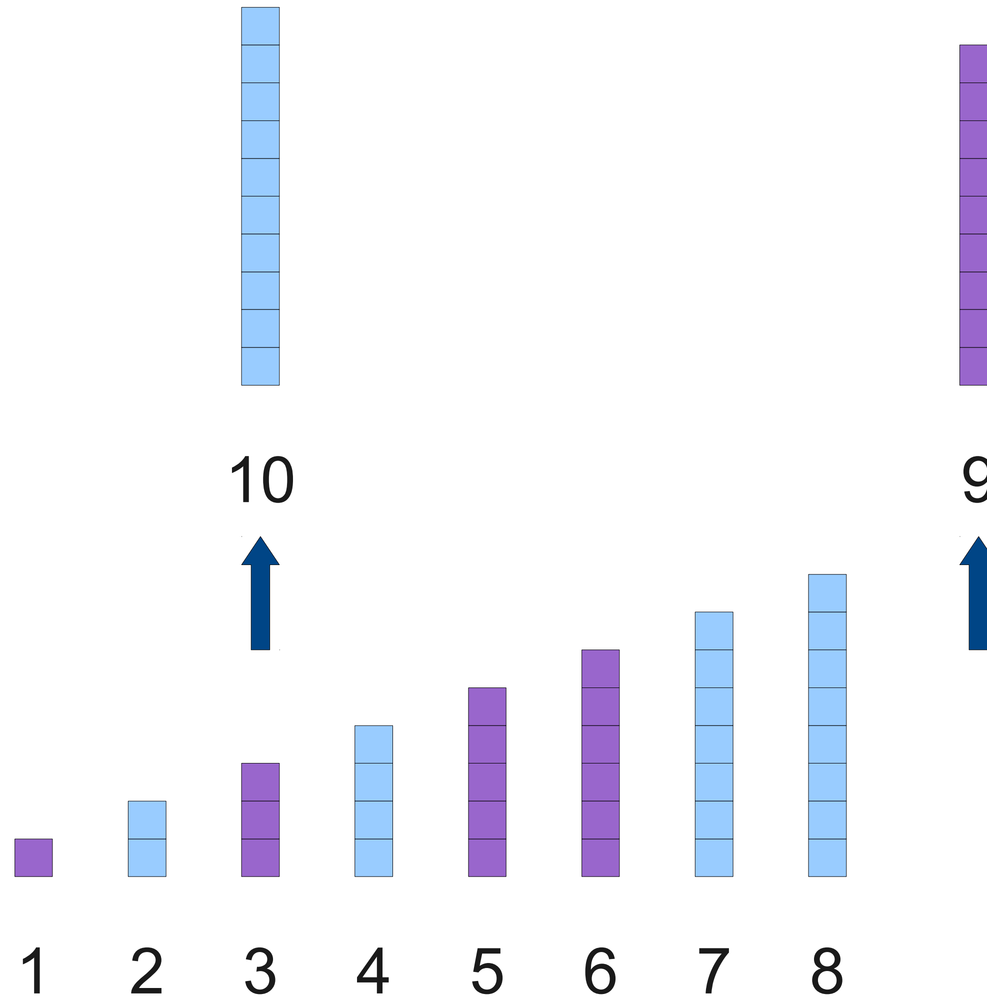
The Key Insight: **Merge**



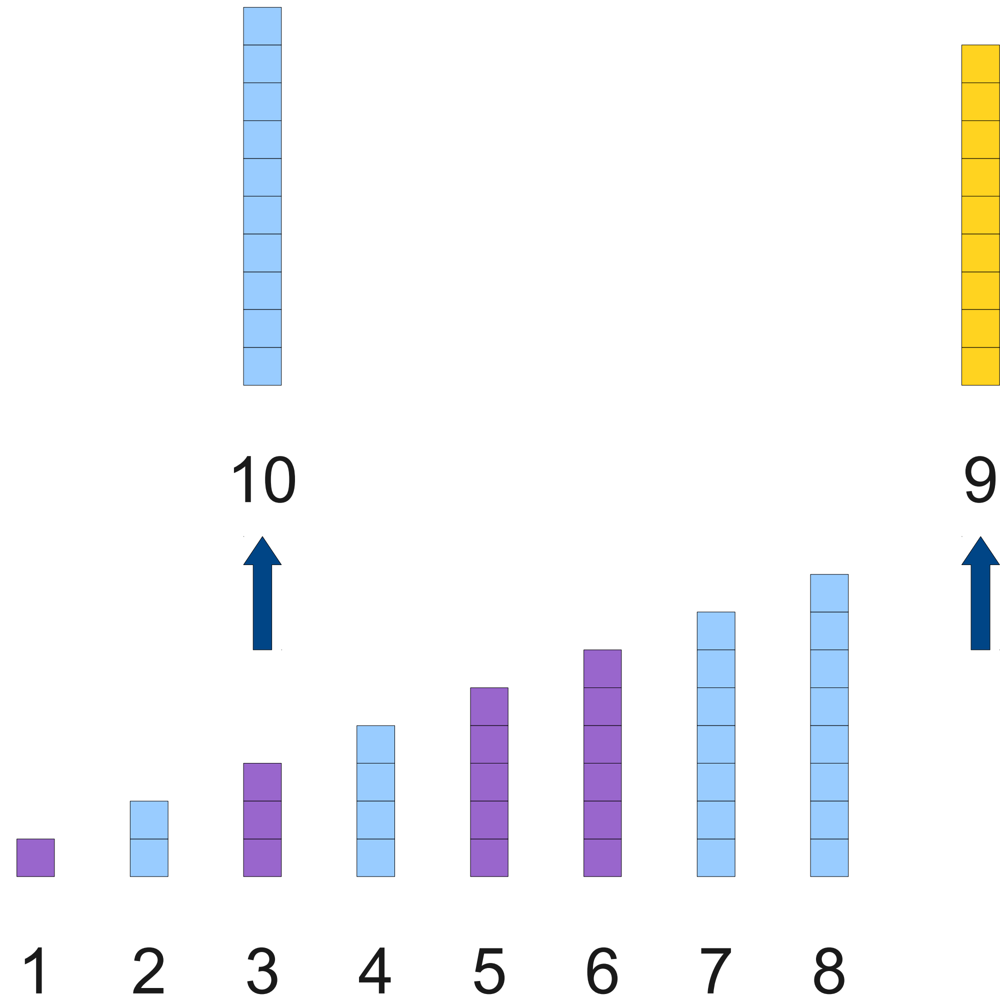
The Key Insight: **Merge**



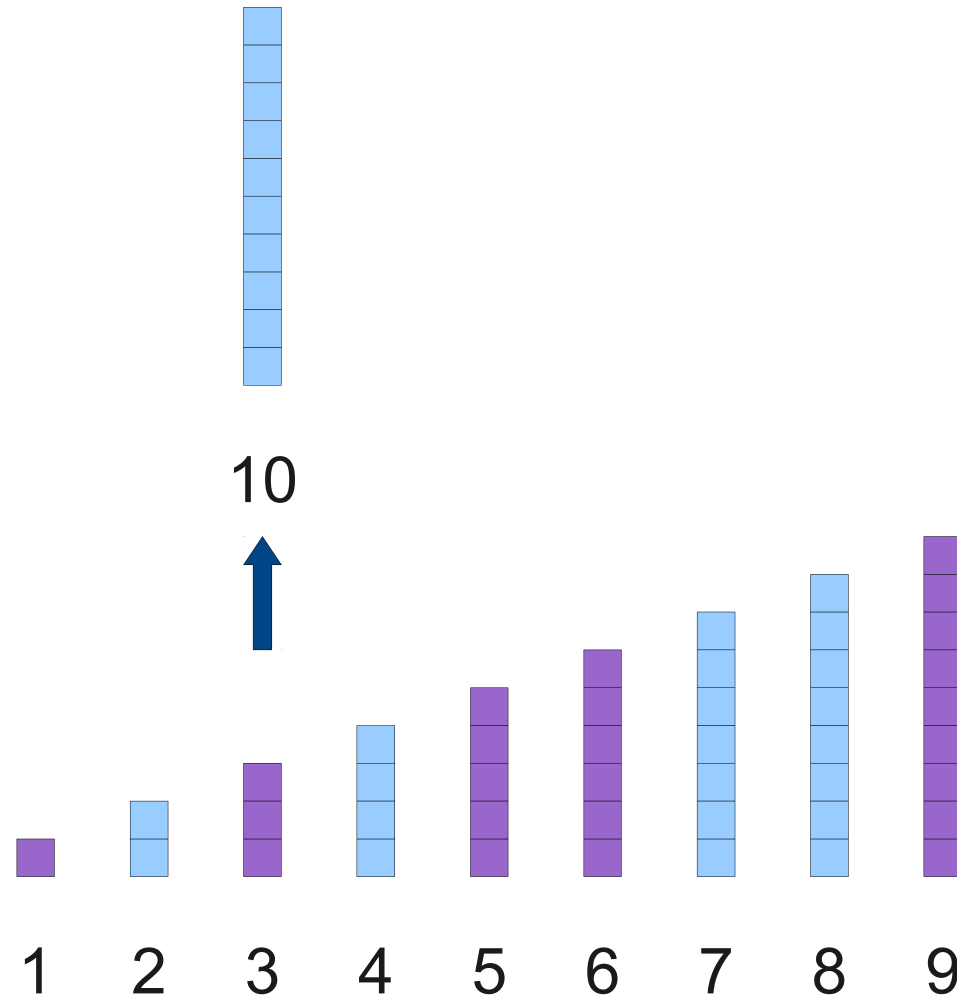
The Key Insight: **Merge**



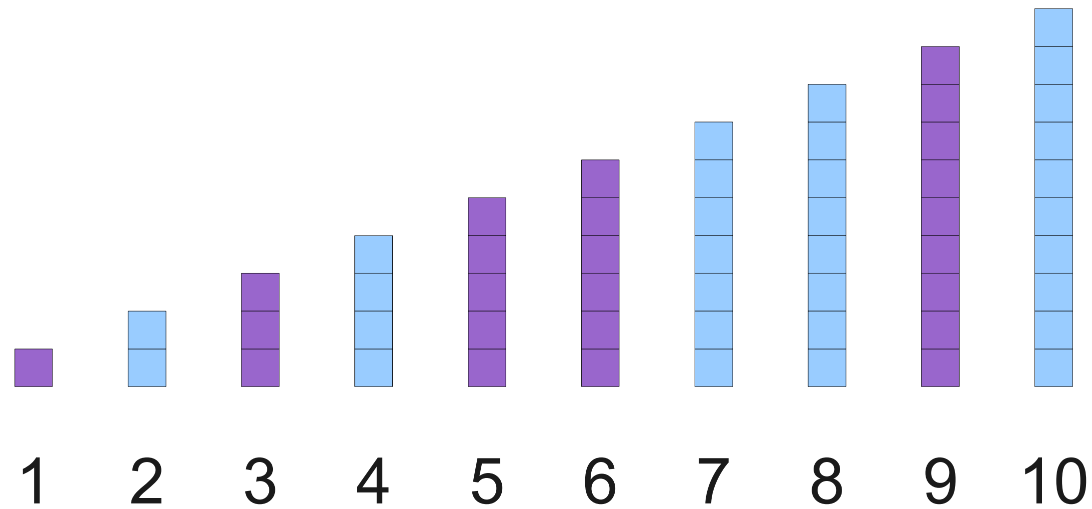
The Key Insight: **Merge**



The Key Insight: **Merge**



The Key Insight: **Merge**



“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);
}
```

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

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*.

Code for Mergesort

Code for 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);
}
```

Code for 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);
}
```

Code for 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);
}
```

Code for 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);
}
```

Code for 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?

Code for 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);
}
```

Code for 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);
}
```

Code for 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);
}
```

Code for Mergesort

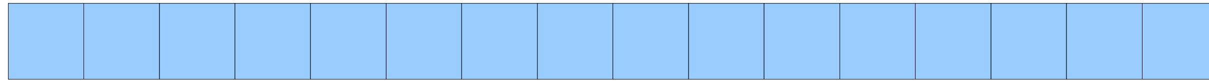
```
void Mergesort(Vector<int>& v) {
   /* Base case: 0- or 1-element lists are already sorted. */
   if (v.size() <= 1) return;

   /* 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);
}
```

Code for Mergesort

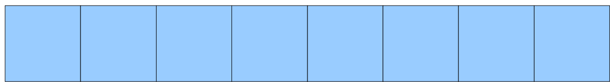
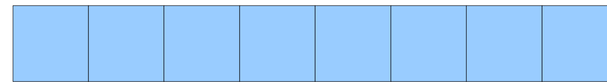
```
void Mergesort(Vector<int>& v) {
   /* Base case: 0- or 1-element lists are already sorted. */
   if (v.size() <= 1) return;

   /* 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);
}
```

How do we analyze this step?

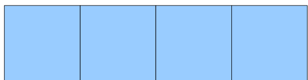
A Graphical Intuition

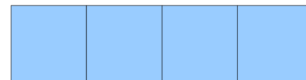
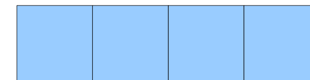


O(n)

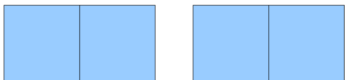
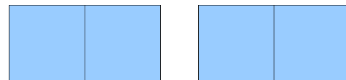
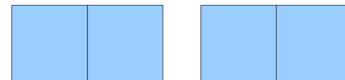
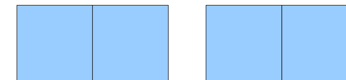



O(n)

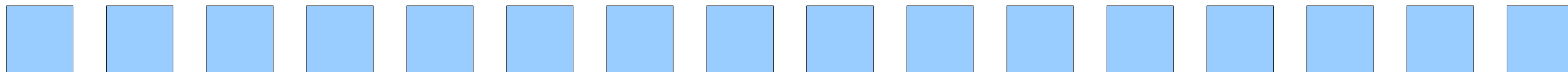


O(n)

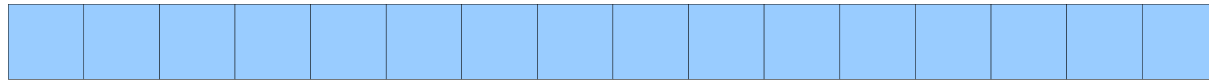





O(n)

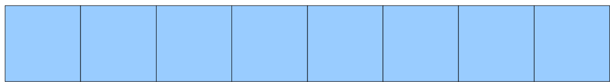


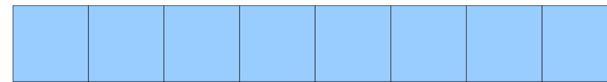
O(n)

A Graphical Intuition

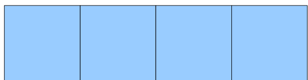


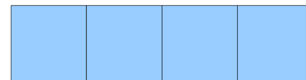
O(n)

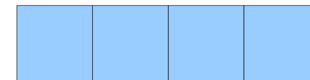




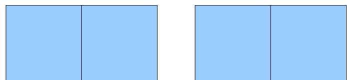
O(n)

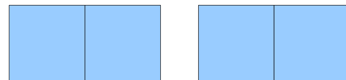


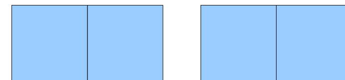


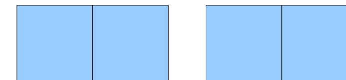


O(n)

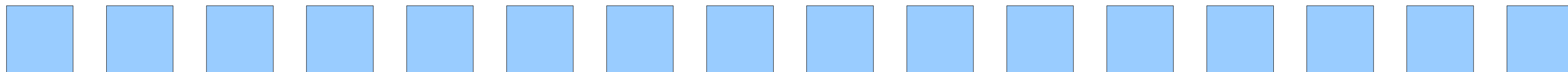








O(n)



O(n)

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:

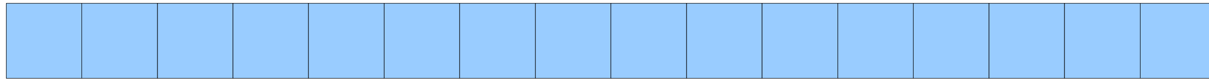
$$N / 2^k = 1$$

$$N = 2^k$$

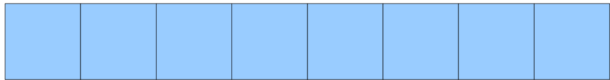
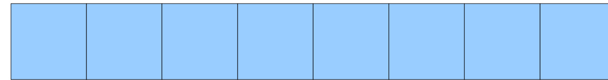
$$\log_2 N = k$$

- So mergesort recurses $\mathbf{\log_2 N}$ levels deep.

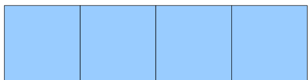
A Graphical Intuition

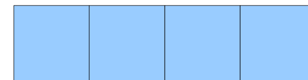


O(n)

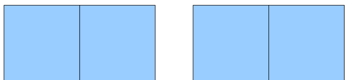
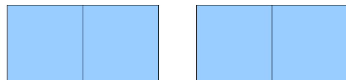
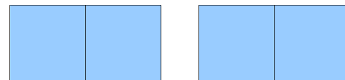
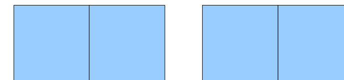



O(n)

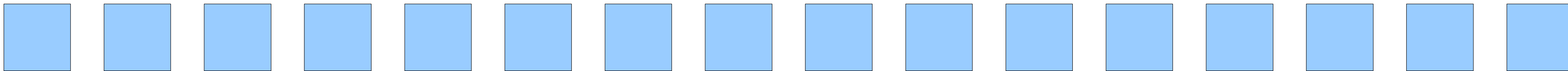




O(n)

O(n)



O(n)

O(n log n)

Mergesort Times

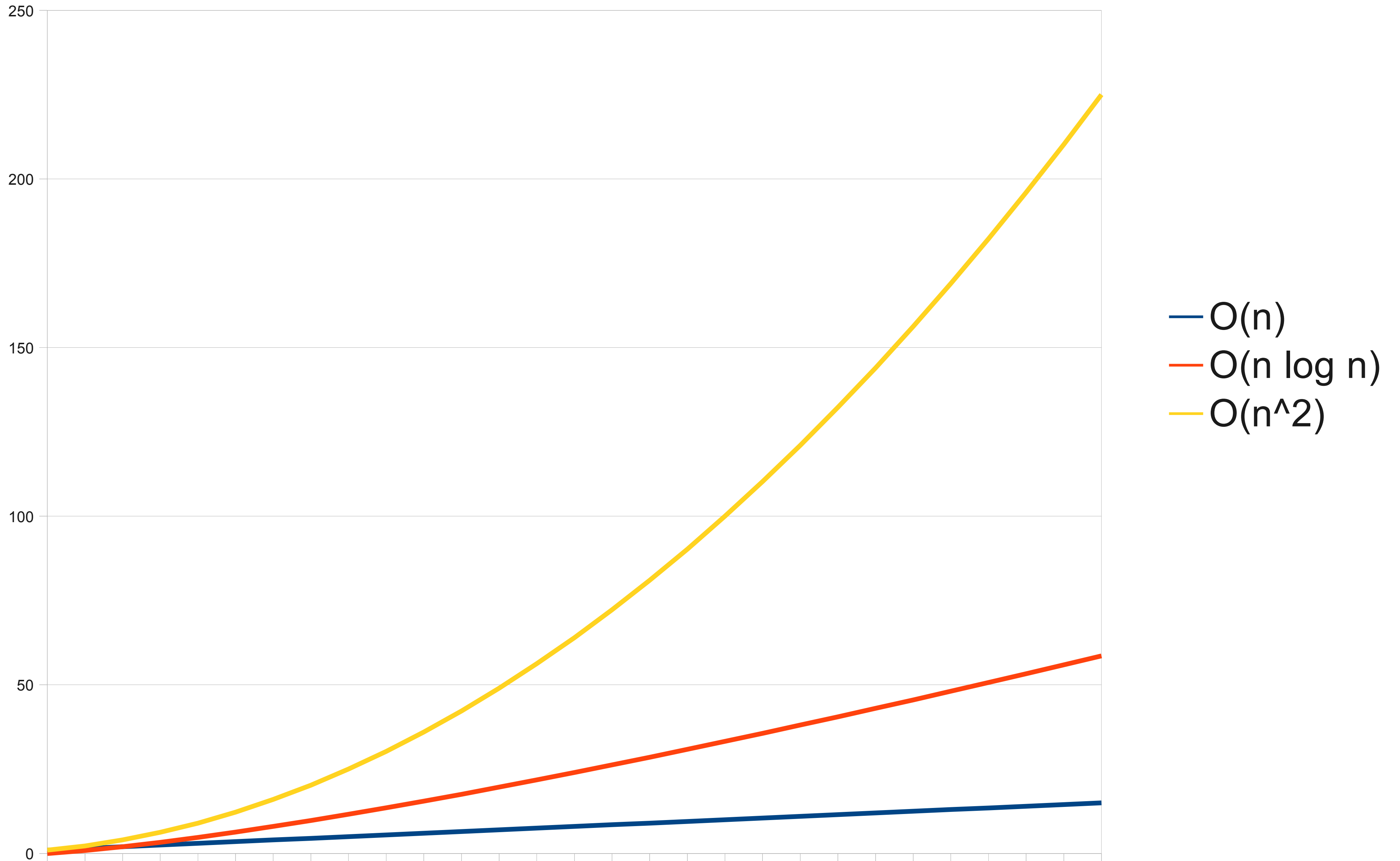
Size	Selection Sort	Insertion Sort	“Split Sort”
10000	0.304	0.160	0.161
20000	1.218	0.630	0.387
30000	2.790	1.427	0.726
40000	4.646	2.520	1.285
50000	7.395	4.181	2.719
60000	10.584	5.635	2.897
70000	14.149	8.143	3.939
80000	18.674	10.333	5.079
90000	23.165	12.832	6.375

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

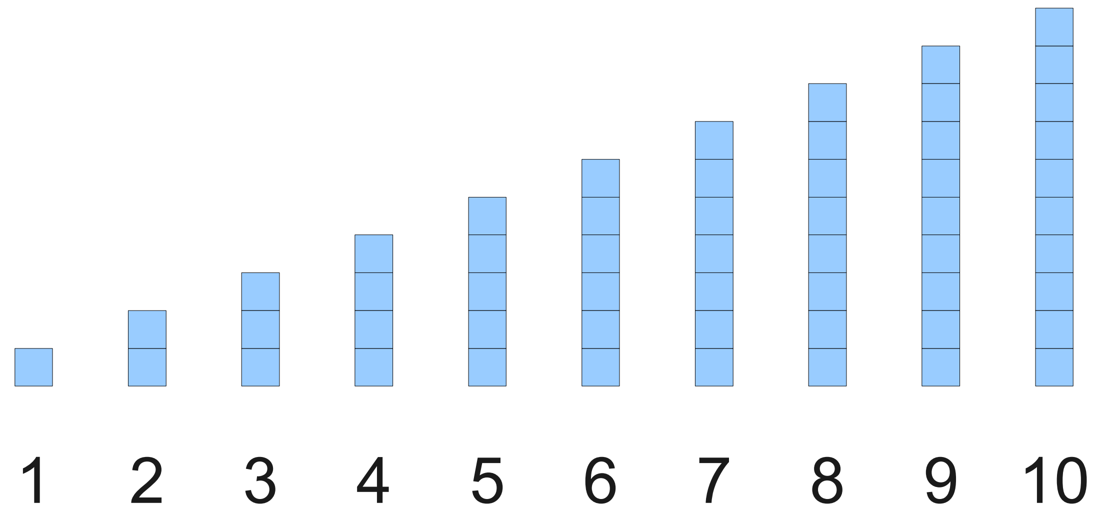
Analysis of Mergesort

- 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).
- **We can only get faster by a constant factor!**

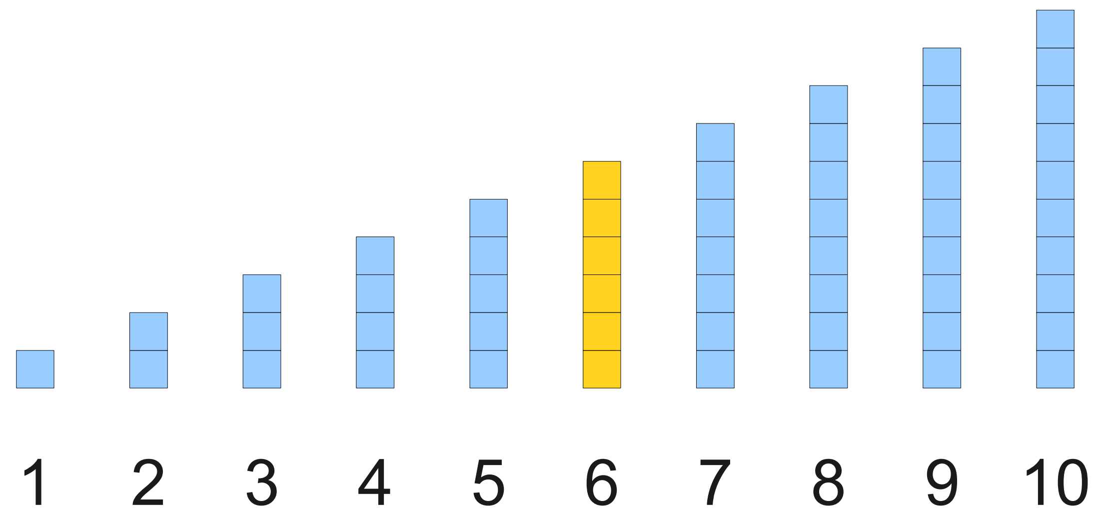

Growth Rates
250
200
150
100
50
0
O(n)
O(n log n)
O(n^2)

A Trivial Observation

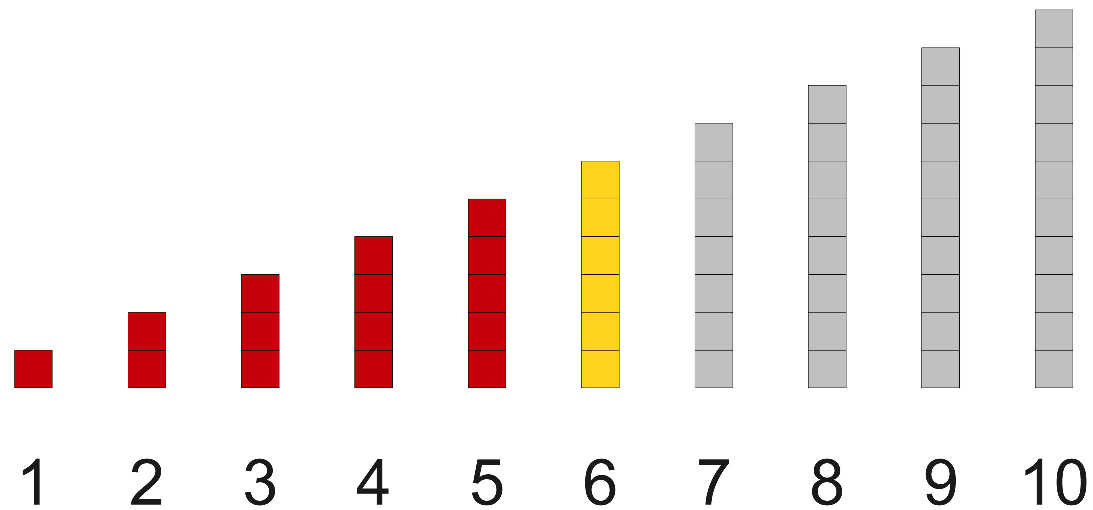
A Trivial Observation



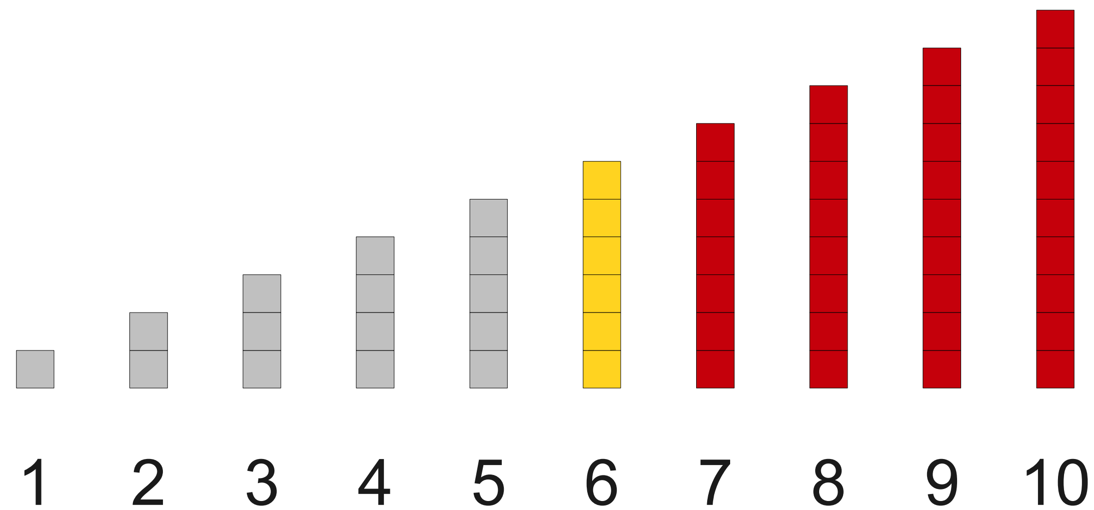
A Trivial Observation



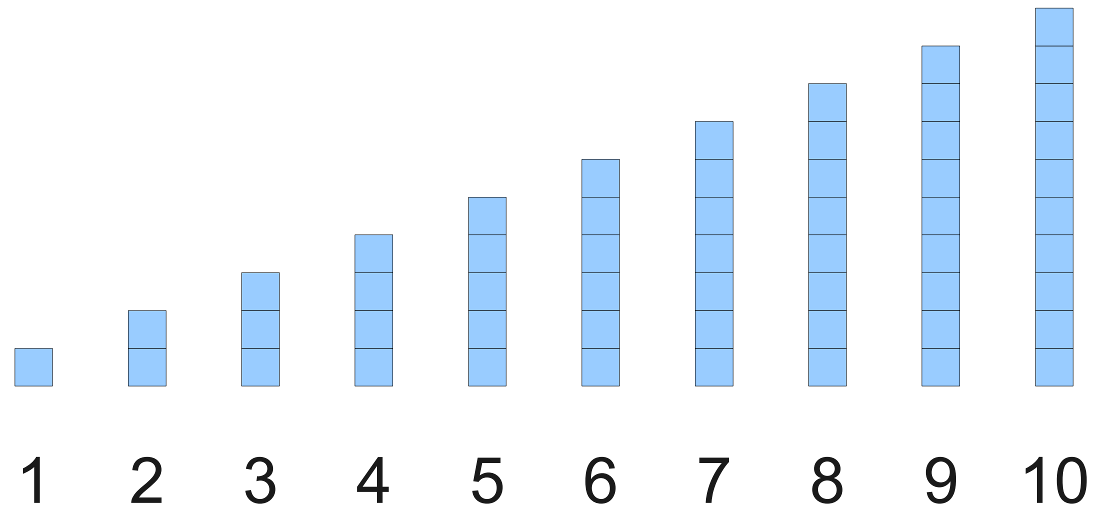
A Trivial Observation



A Trivial Observation



A Trivial Observation



So What?

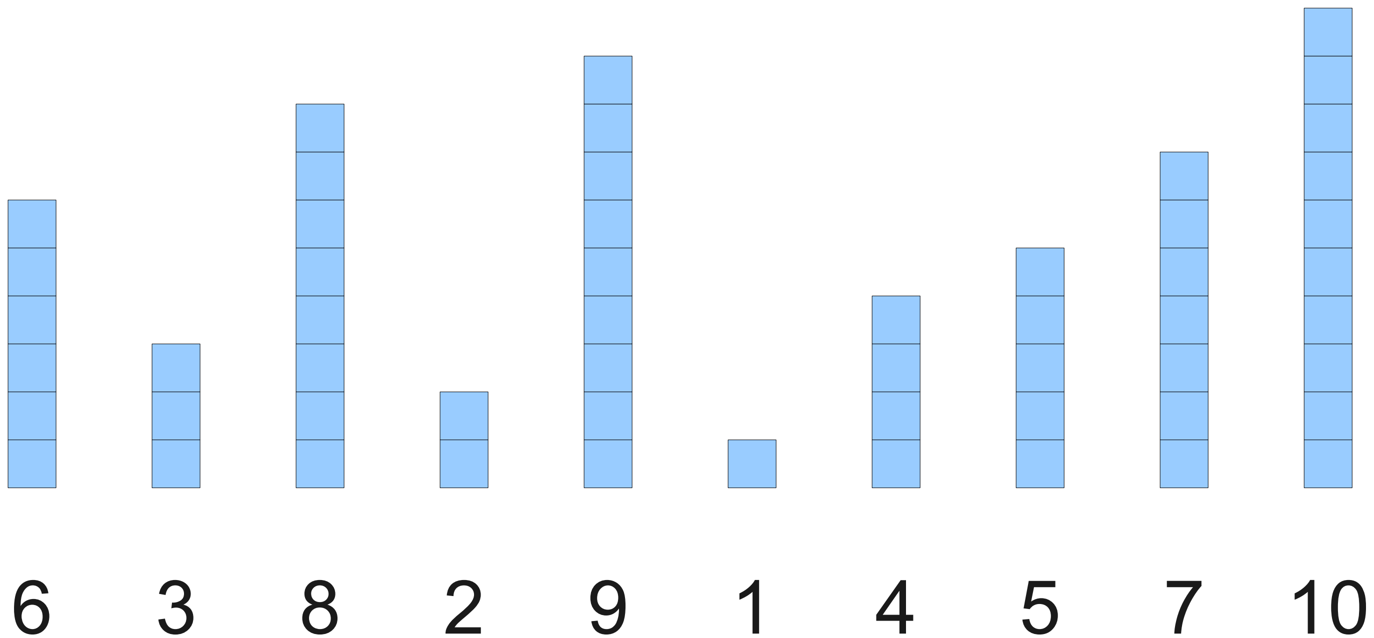
- This idea leads to a particularly clever sorting algorithm.
- Idea:
  - Pick an element from the array.
  - Put the smaller elements on one side.
  - Put the bigger elements on the other side.
  - Recursively sort each half.
- But how do we do the middle two steps?

Partitioning

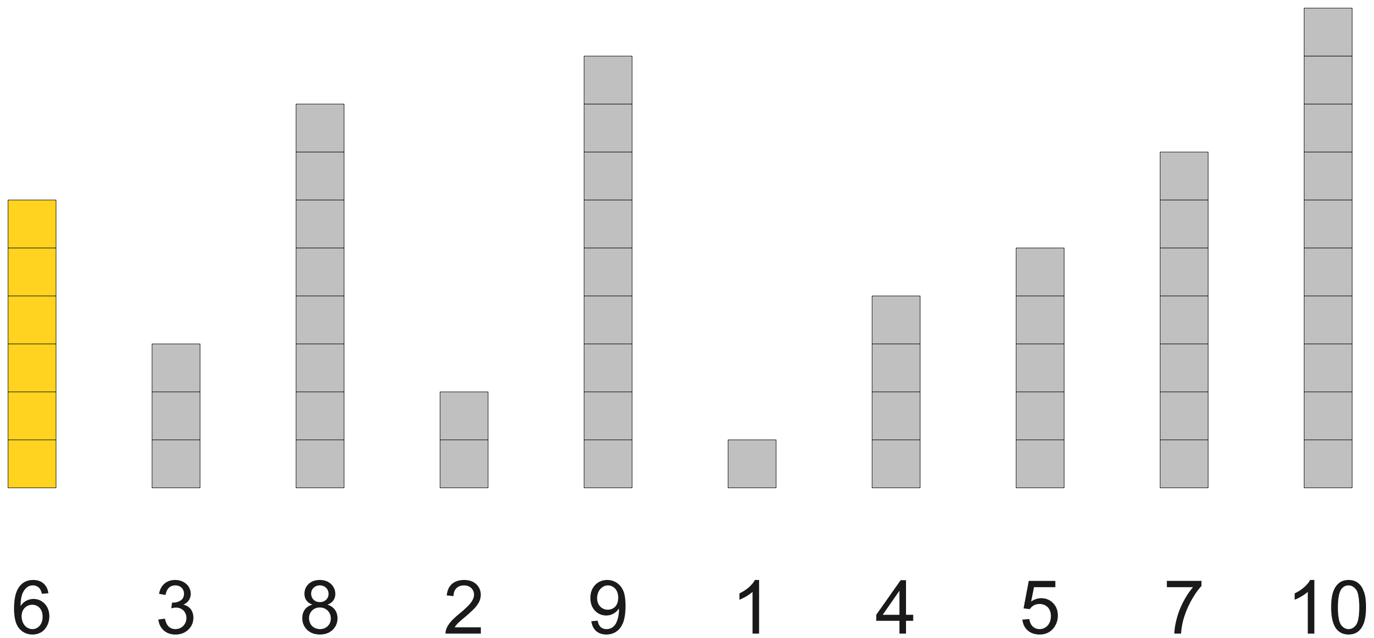
- Pick a **pivot element**.
- Move everything less than the pivot to the left of the pivot.
- Move everything greater than the pivot to the right of the pivot.
- Good news: O(n) algorithm exists!
- Bad news: it's a bit tricky...

The Partition Algorithm

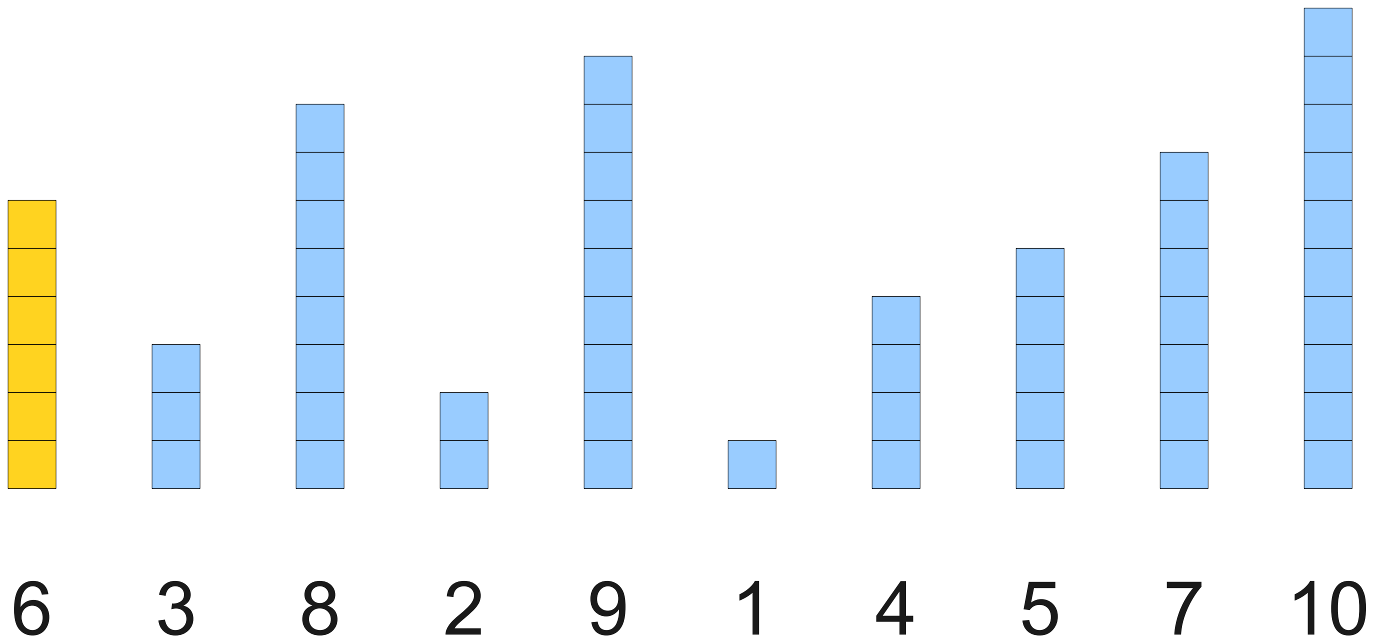
The Partition Algorithm



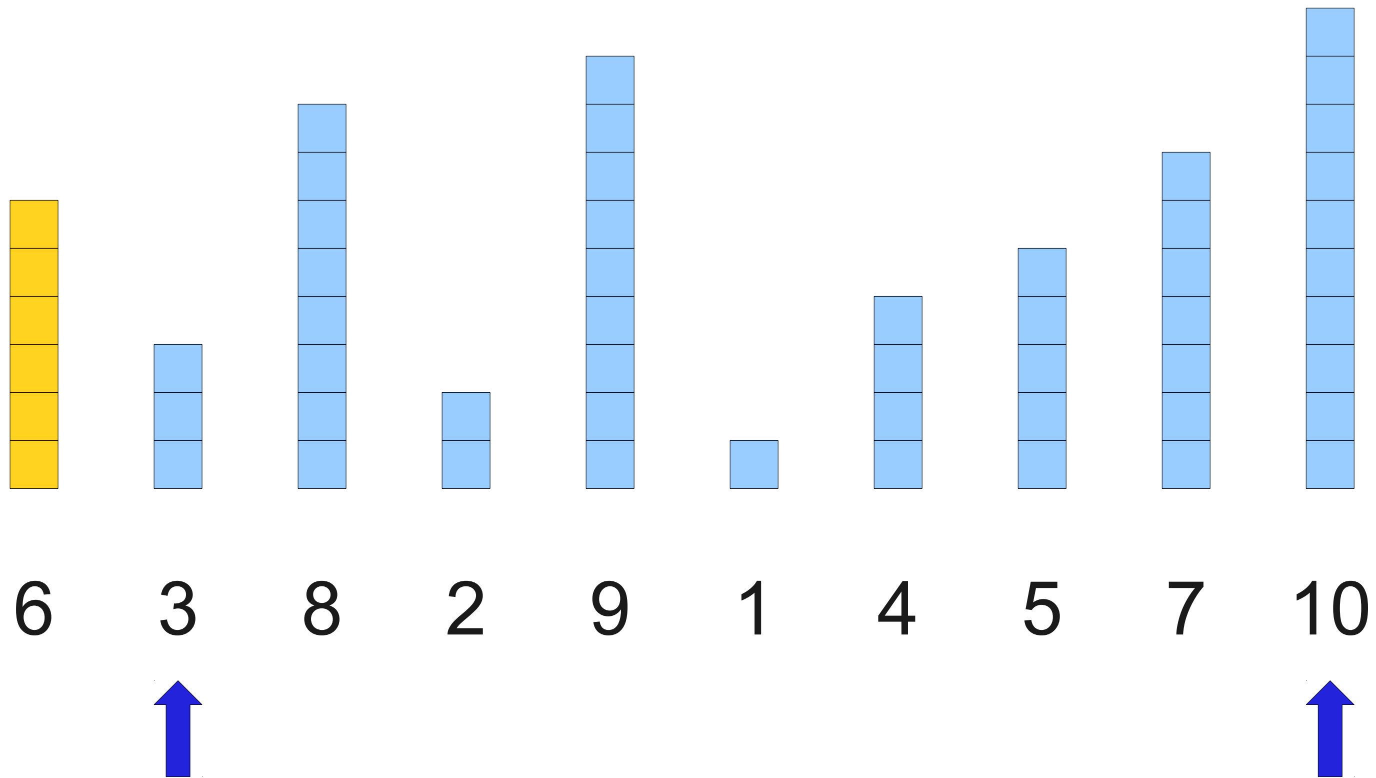
The Partition Algorithm



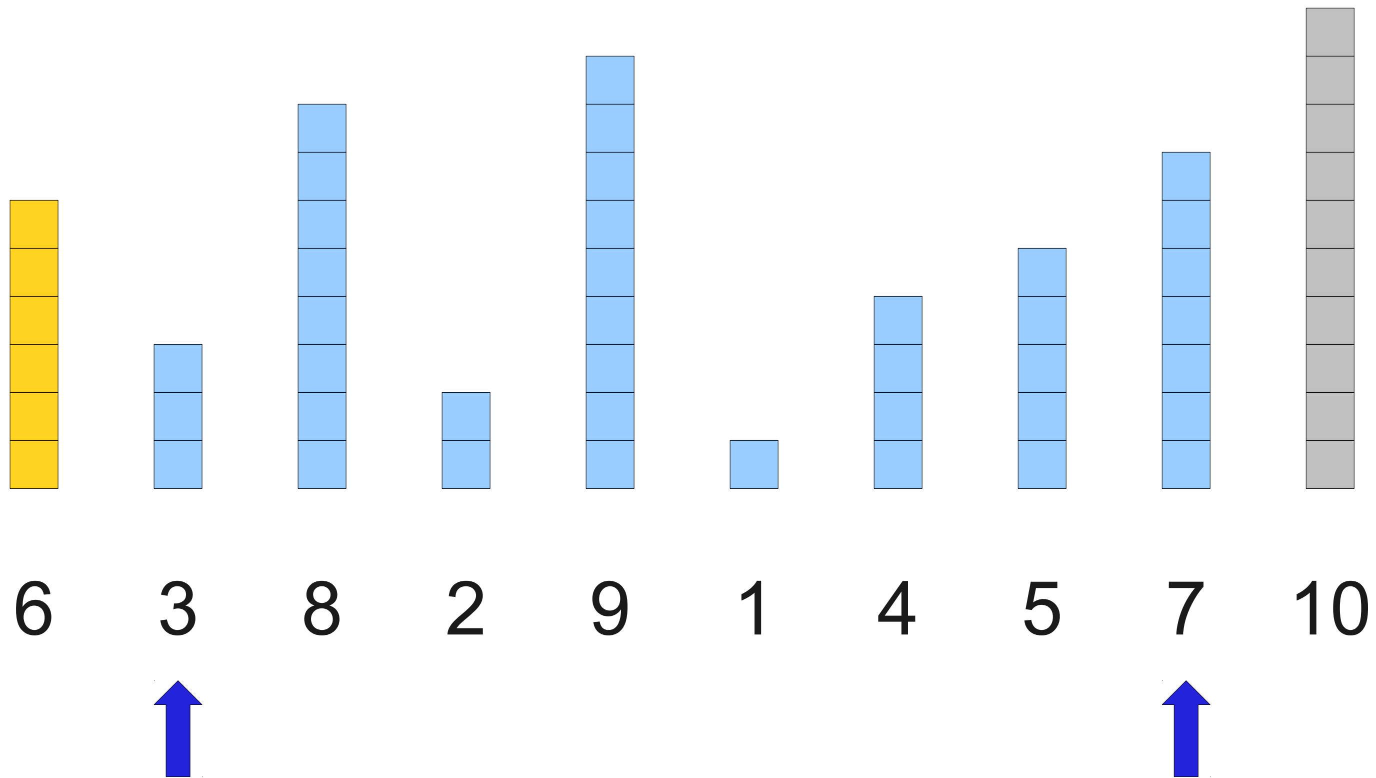
The Partition Algorithm



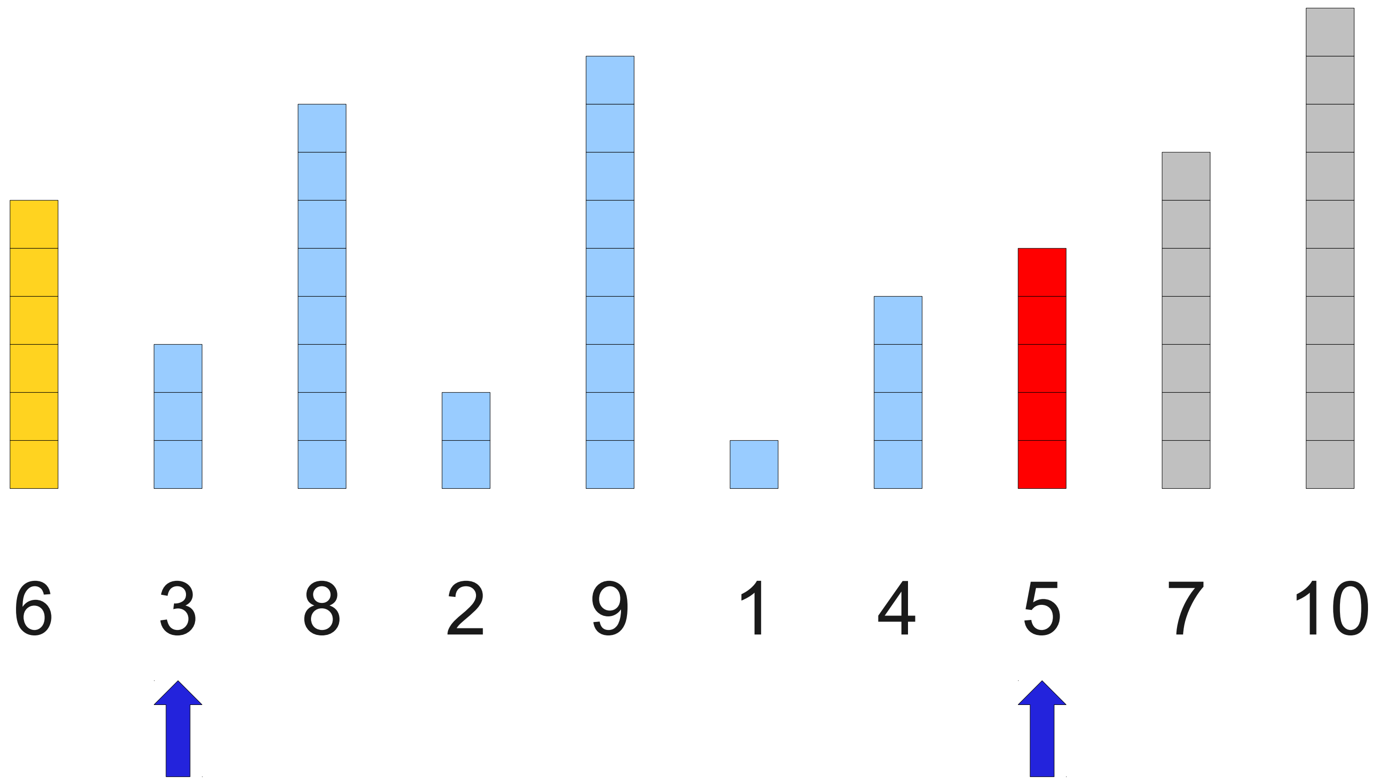
The Partition Algorithm



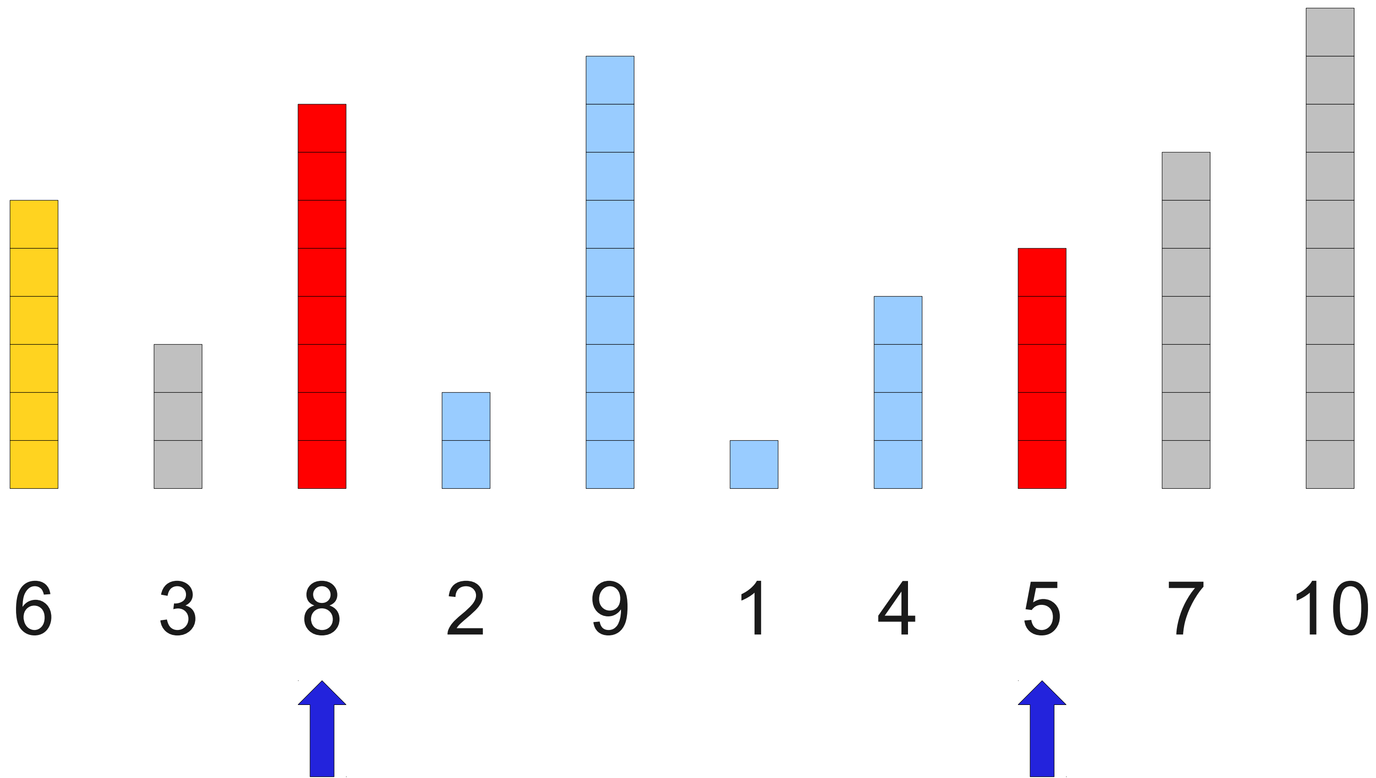
The Partition Algorithm



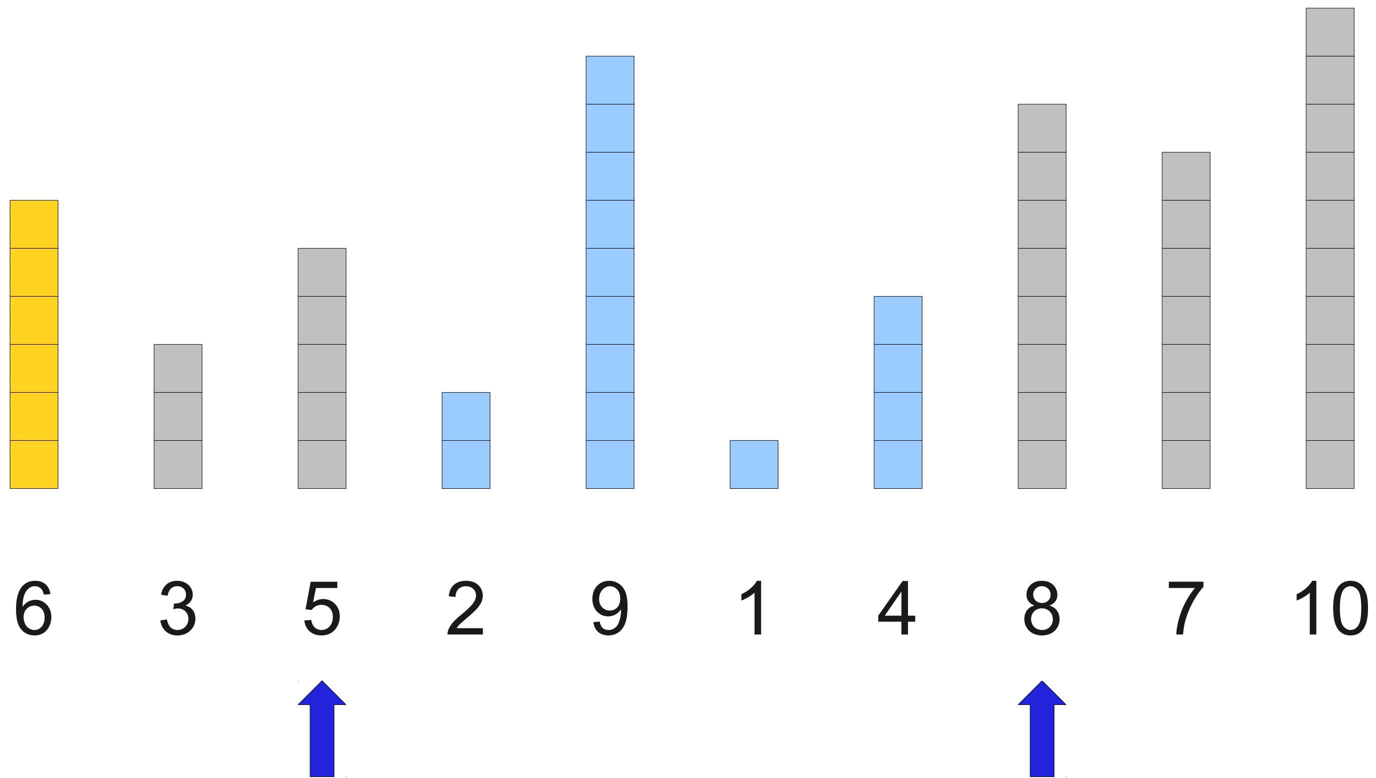
The Partition Algorithm



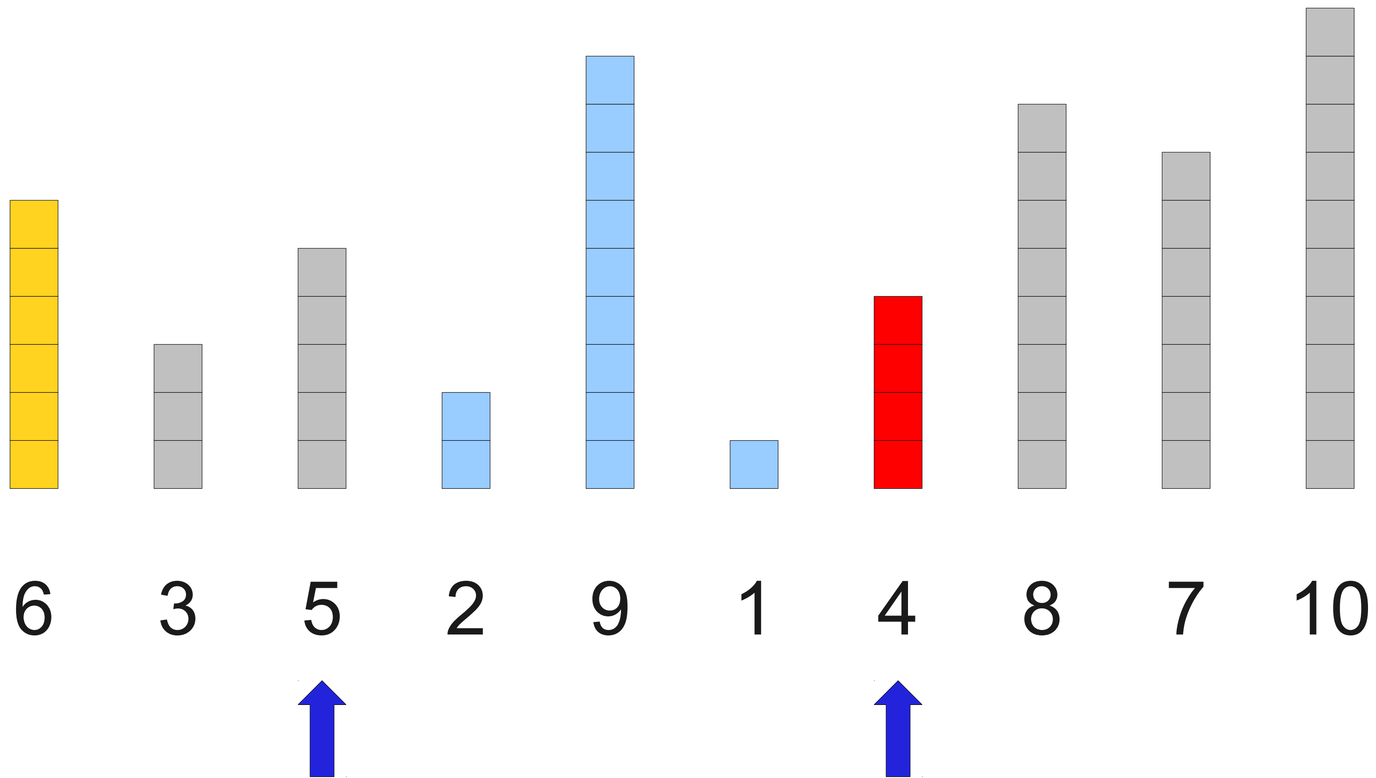
The Partition Algorithm



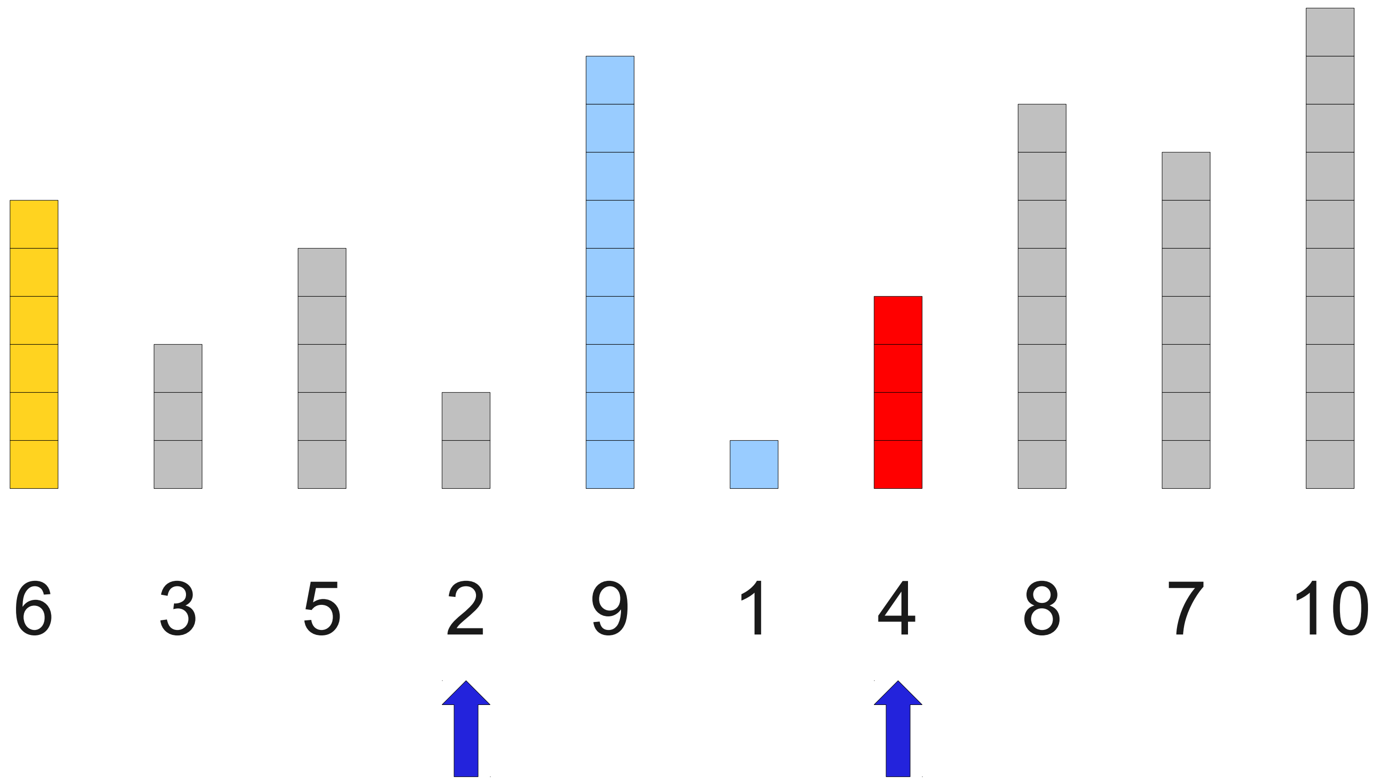
The Partition Algorithm



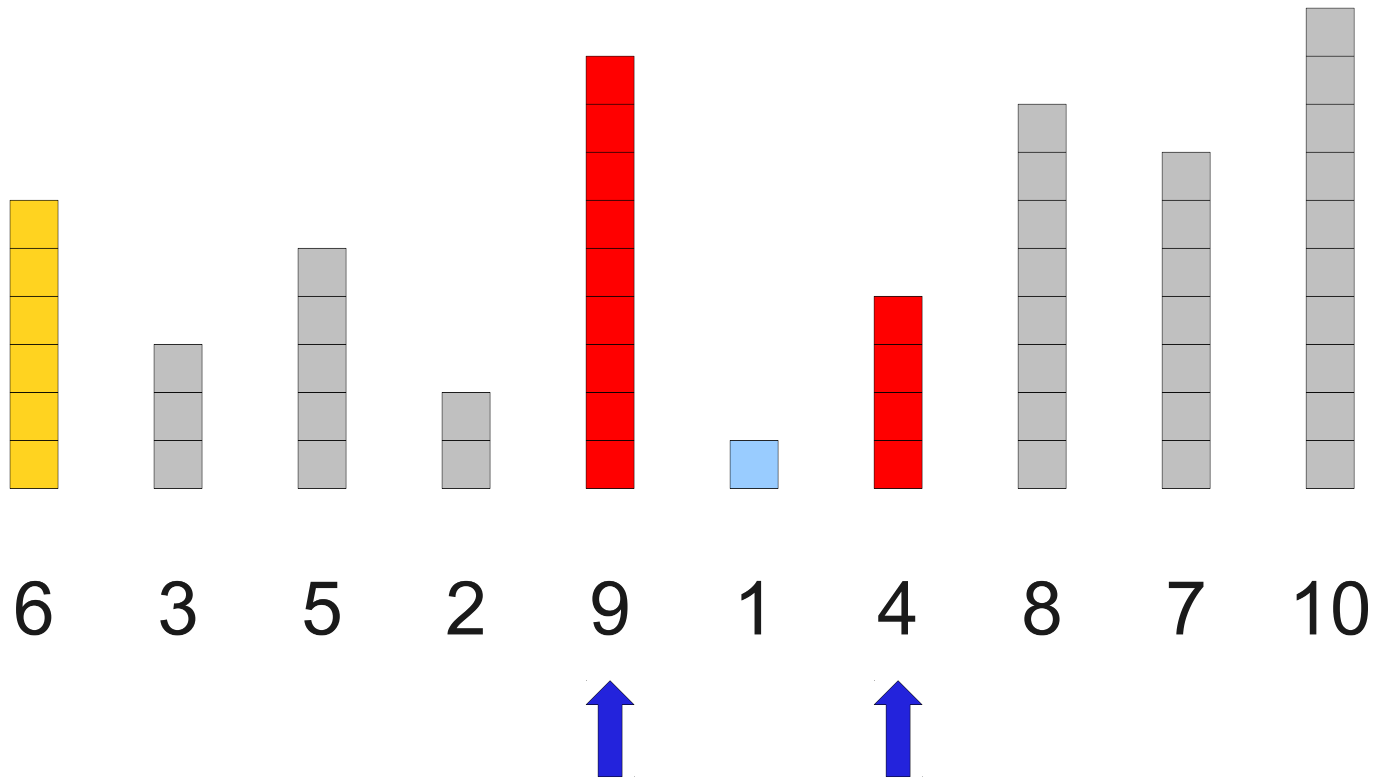
The Partition Algorithm



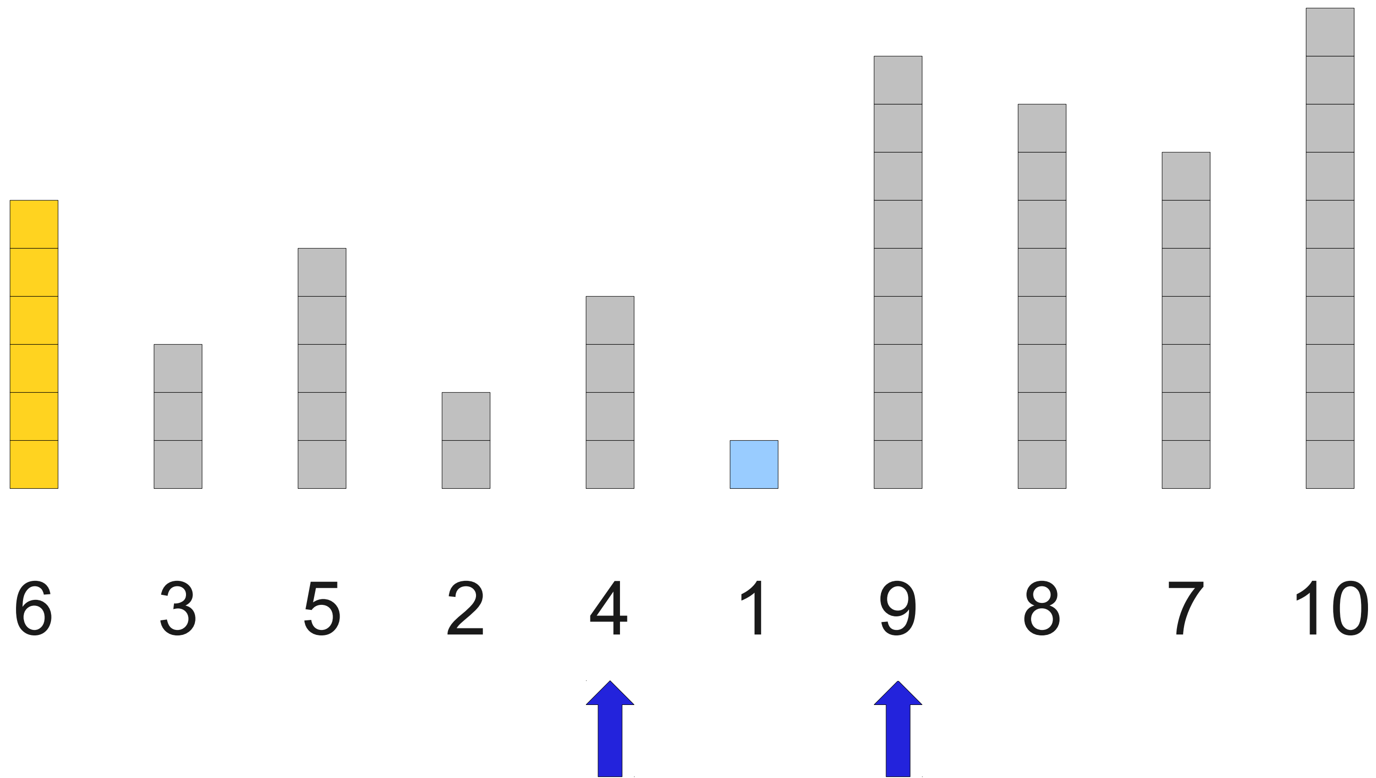
The Partition Algorithm



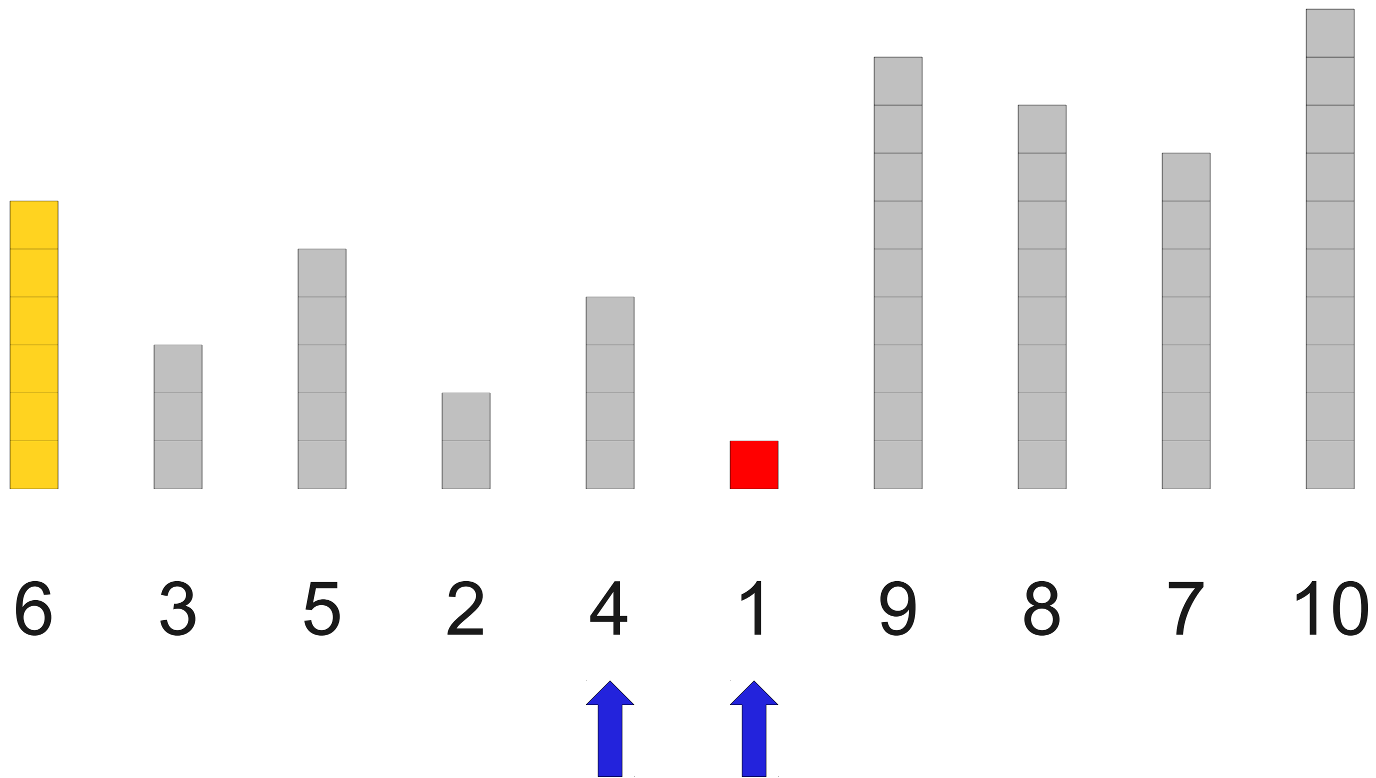
The Partition Algorithm



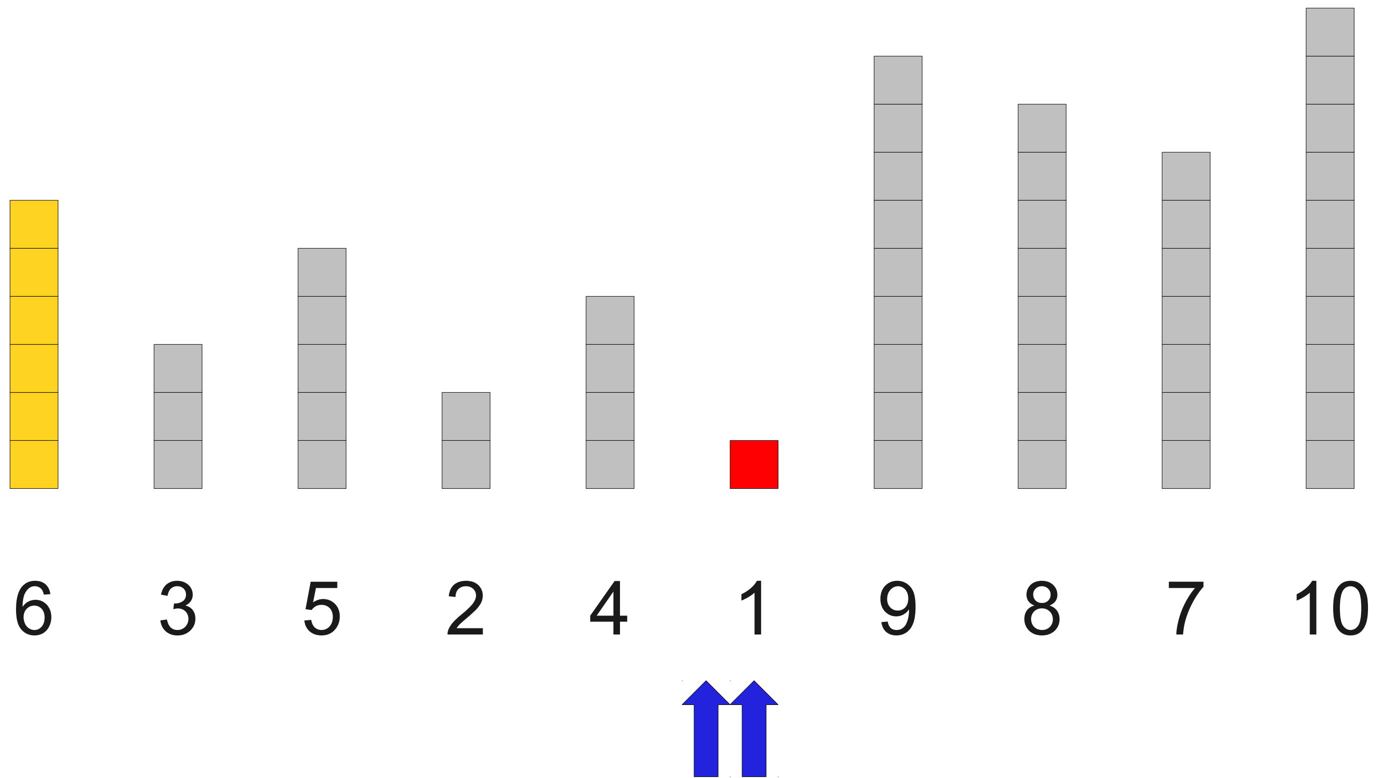
The Partition Algorithm



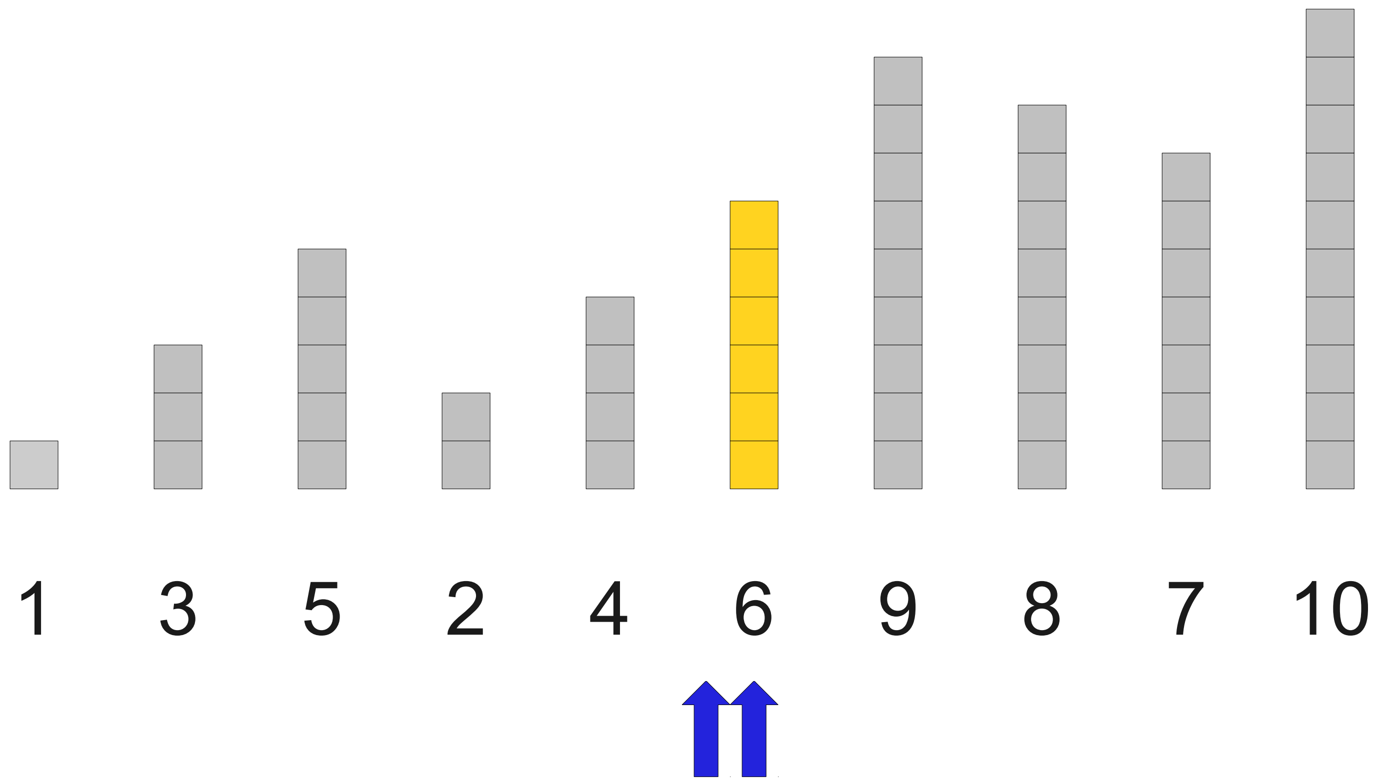
The Partition Algorithm



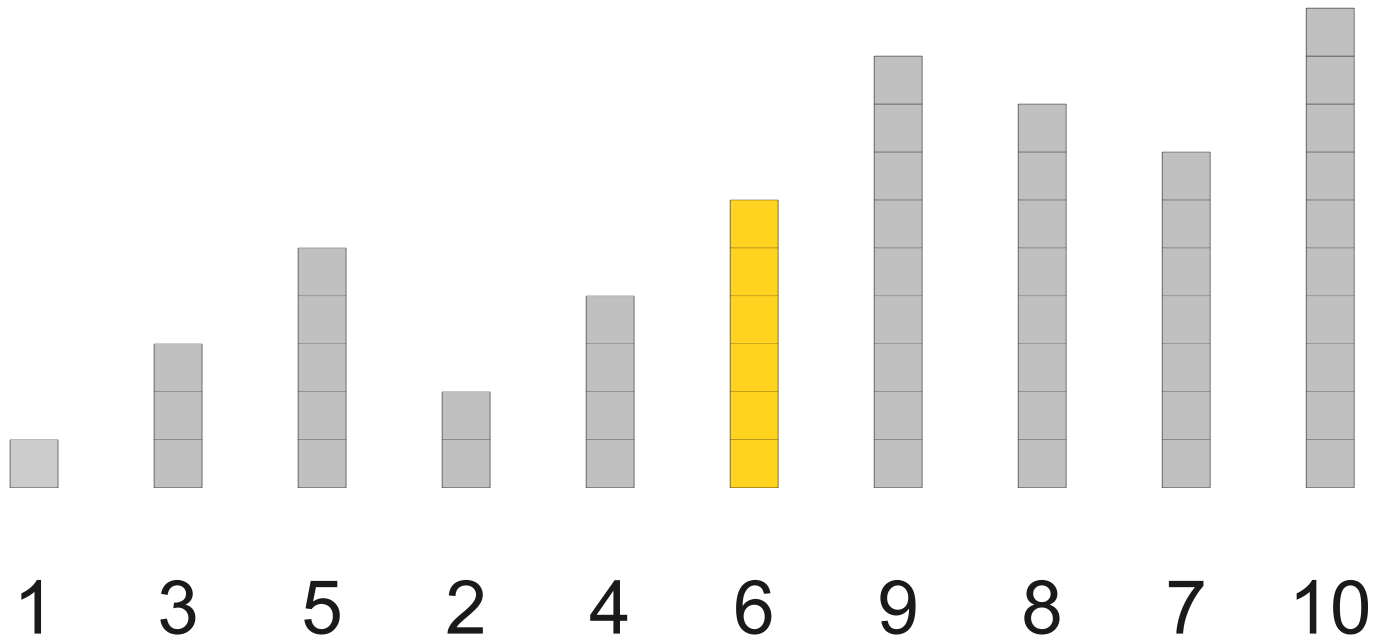
The Partition Algorithm



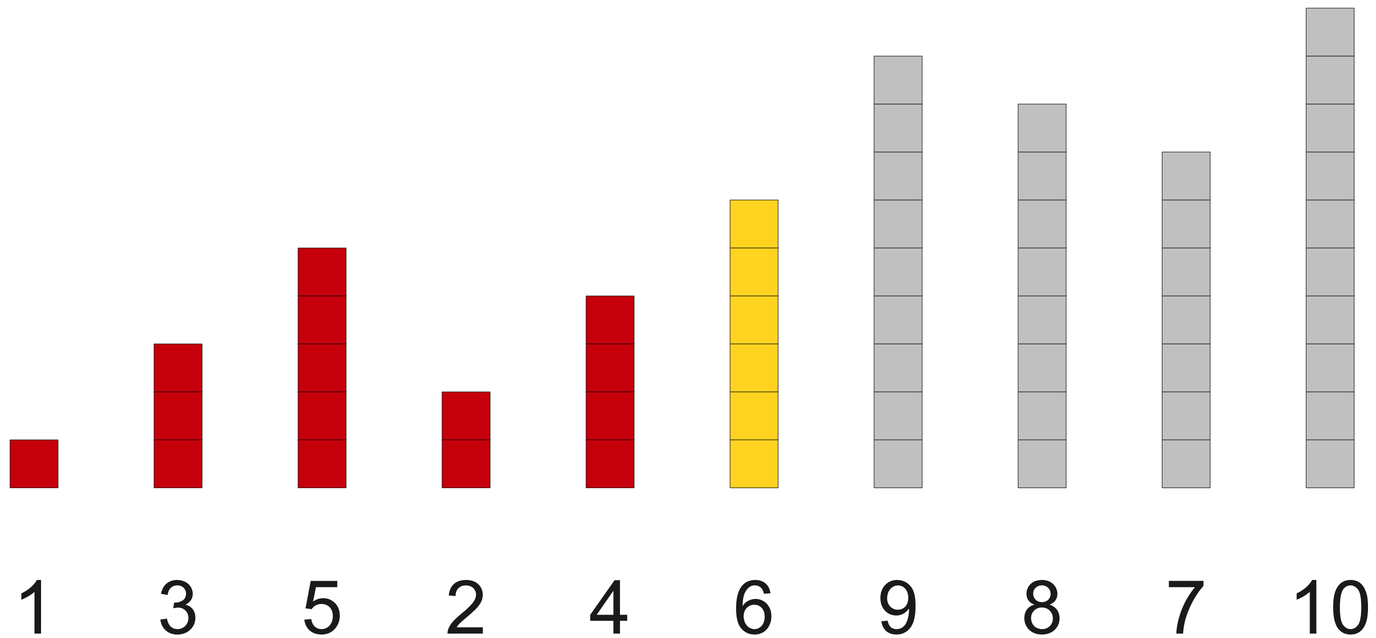
The Partition Algorithm



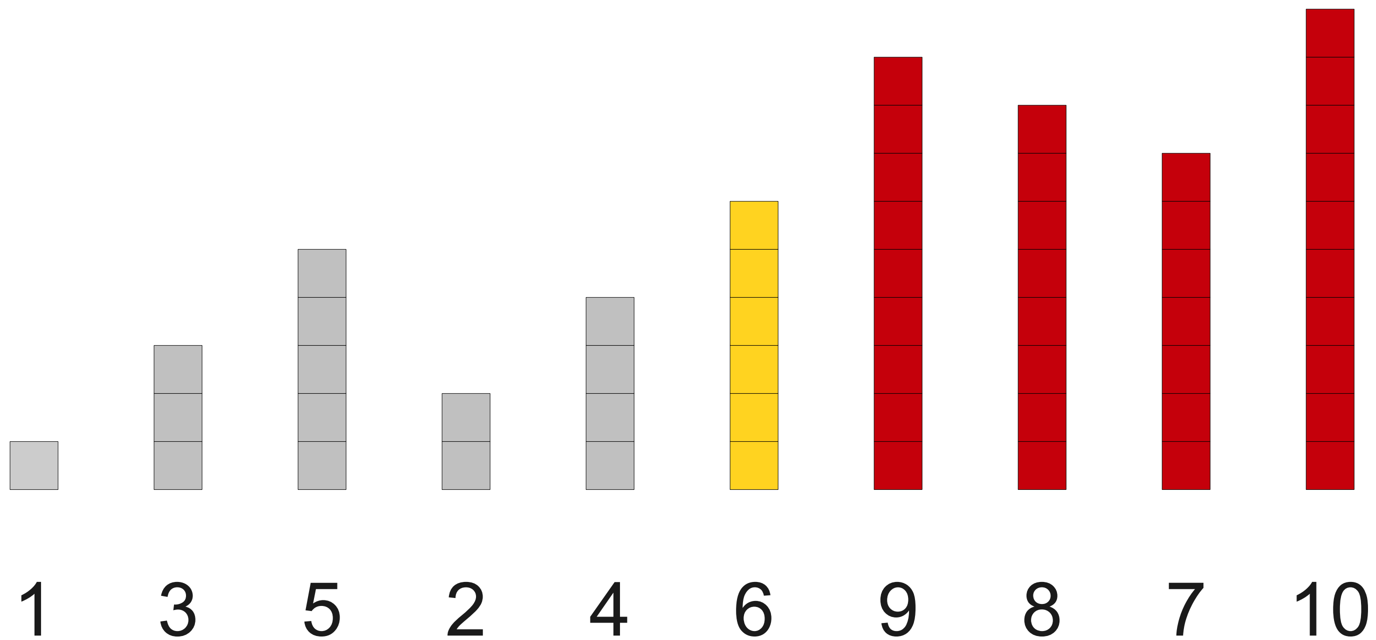
The Partition Algorithm



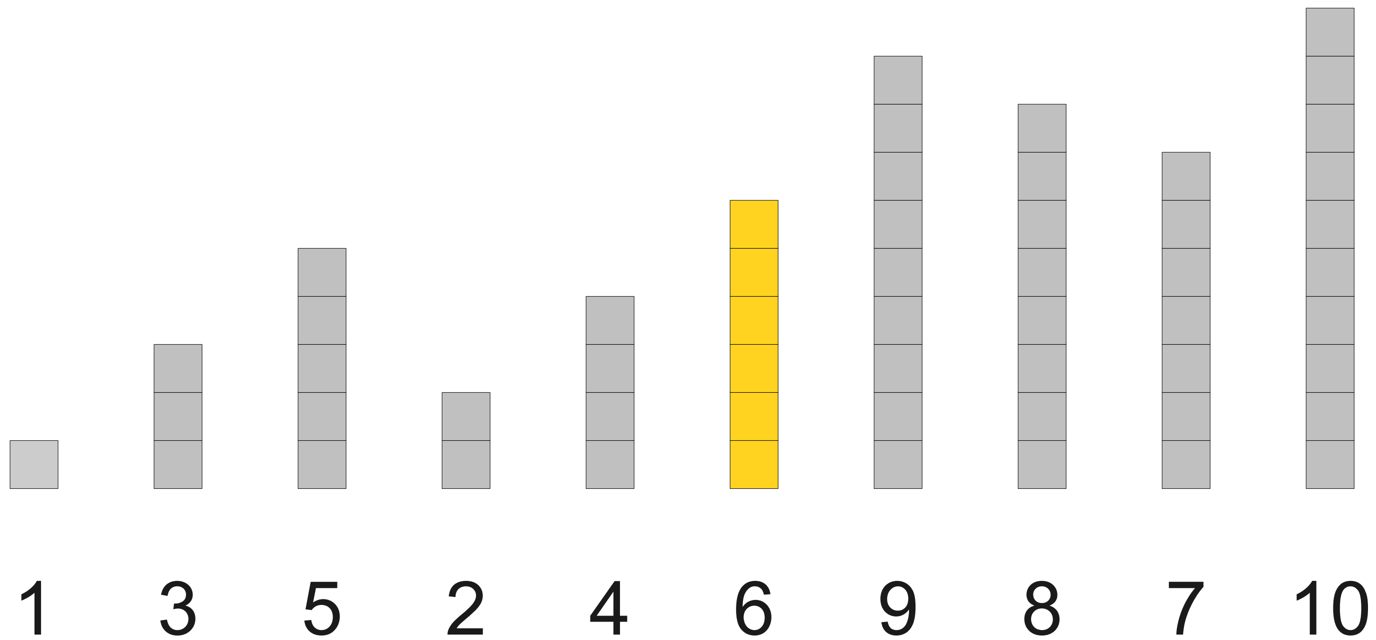
The Partition Algorithm



The Partition Algorithm



The Partition Algorithm



Code for Partition

Code for Partition

```
int Partition(Vector<int>& v, int low, int high) {
    int pivot = v[low];
    int left  = low + 1, right = high;

    while (left < right) {
        while (left < right && v[right] >= pivot) --right;
        while (left < right && v[left]  < pivot) ++left;

        if (left < right)
           swap(v[left], v[right]);
    }
    if (pivot < v[right])
        return low;

    swap (v[low], v[right]);
    return right;
}
```

A Partition-Based Sort

- Idea:
  - Partition the array around some element.
  - Recursively sort the left and right halves.
- This works extremely quickly.
- In fact… the algorithm is called *quicksort*.

Quicksort

Quicksort

```
void Quicksort(Vector<int>& v, int low, int high) {
    if (low >= high) return;

    int partitionPoint = Partition(v, low, high);
    Quicksort(v, low, partitionPoint – 1);
    Quicksort(v, partitionPoint + 1, high);
}
```

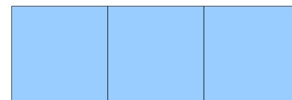
How fast is quicksort?

It depends on our choice of pivot.

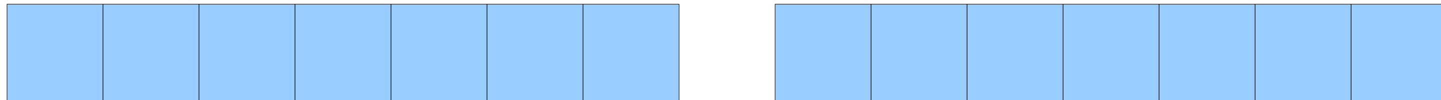
Suppose we get lucky...

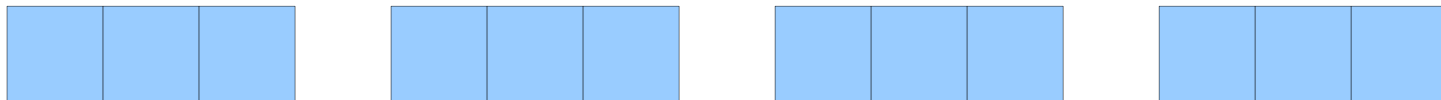
Suppose we get lucky...

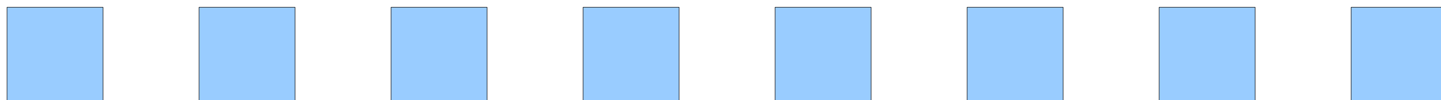
Suppose we get lucky...



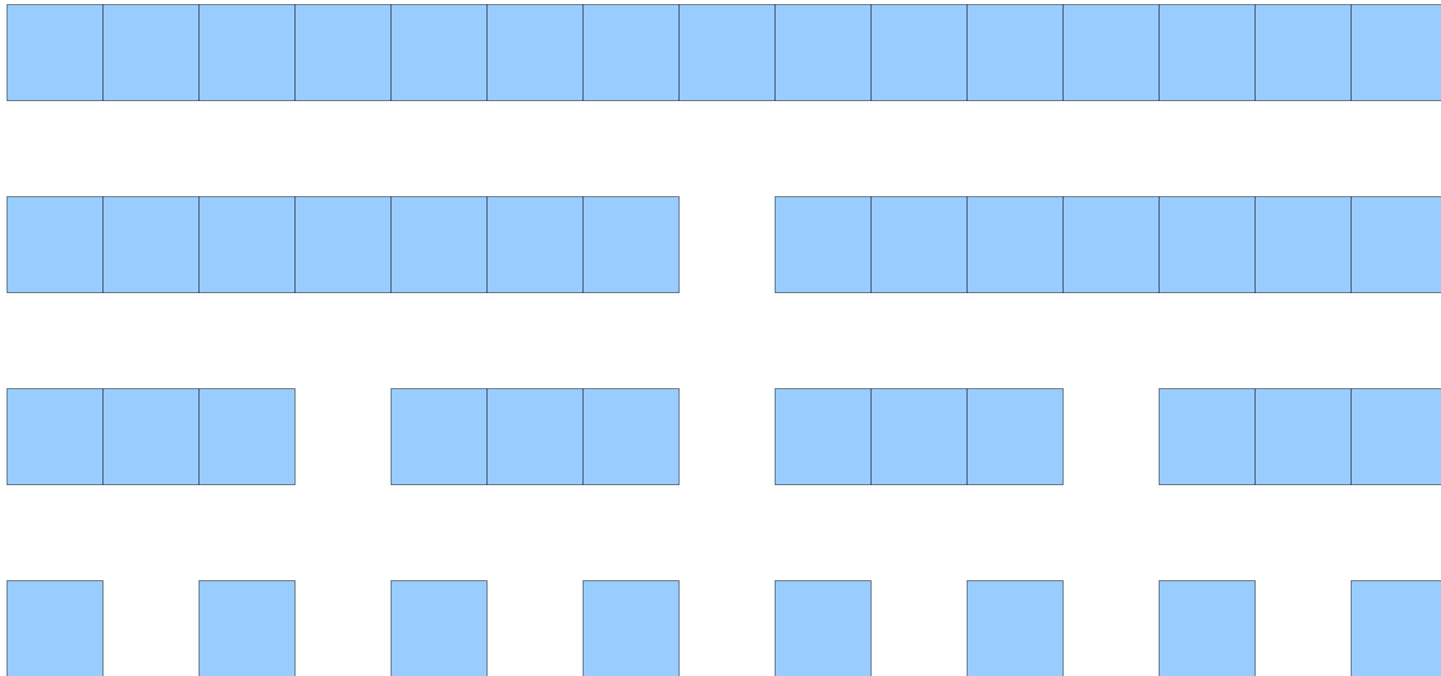
Suppose we get lucky...







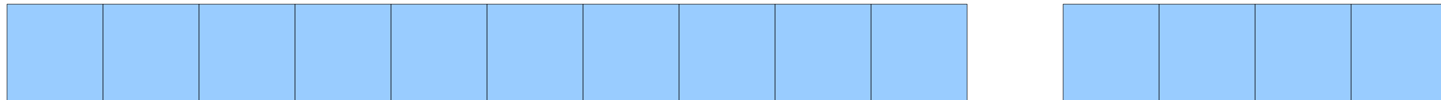
Suppose we get lucky...



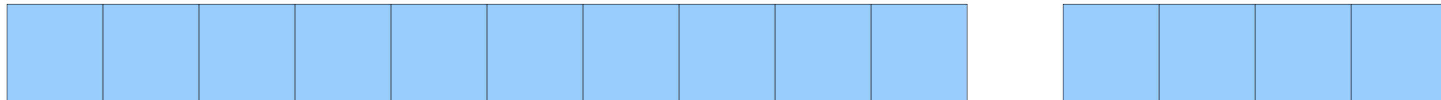
O(n log n)

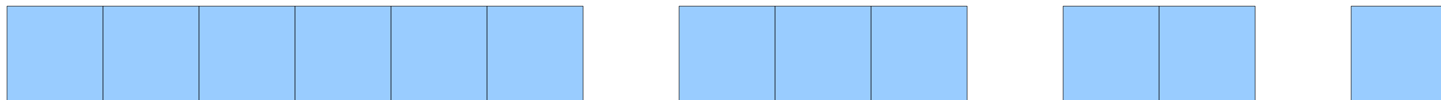
Suppose we get *sorta* lucky...

Suppose we get *sorta* lucky...

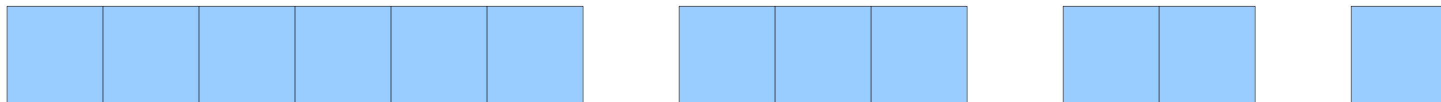


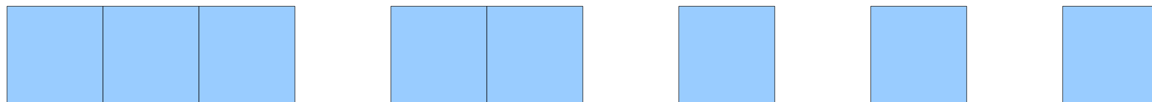
Suppose we get *sorta* lucky...



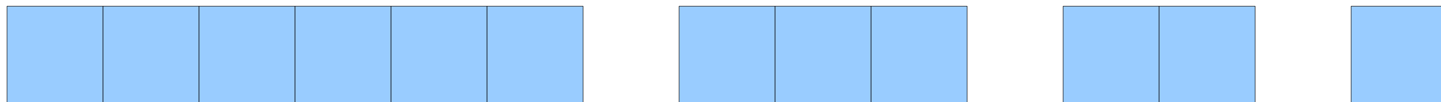


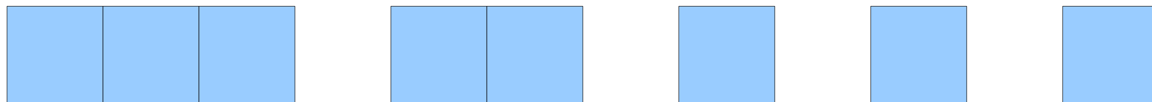
Suppose we get *sorta* lucky...

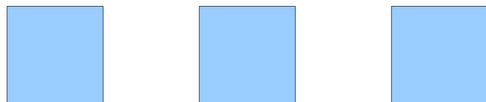




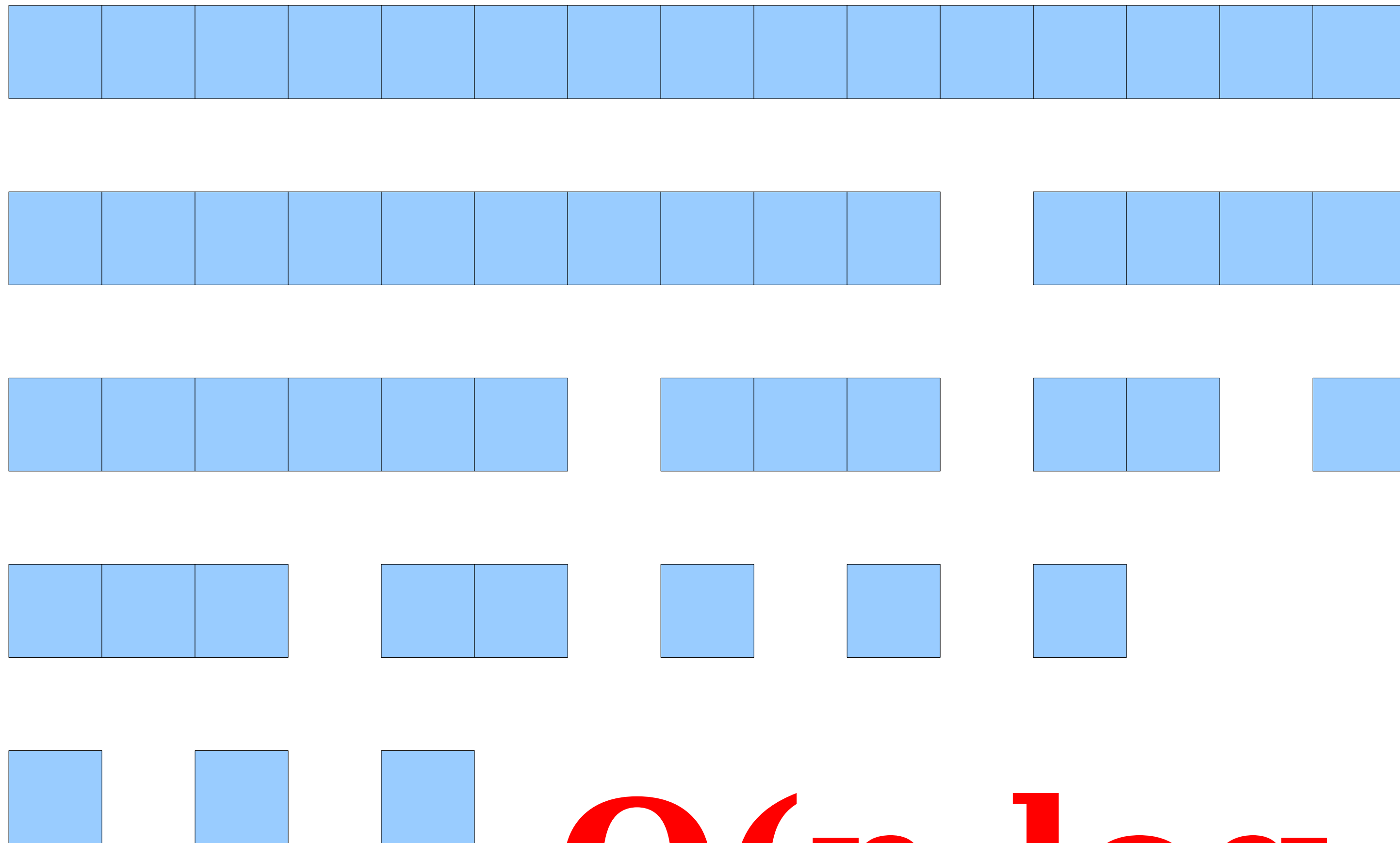
Suppose we get *sorta* lucky...







Suppose we get *sorta* lucky...



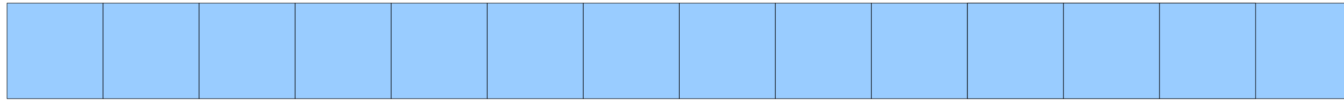
Suppose we get **unlucky**

Suppose we get **unlucky**

Suppose we get **unlucky**

Suppose we get **unlucky**

Suppose we get **unlucky**



...

Suppose we get **unlucky**

n

n - 1

n - 2

n - 3

...

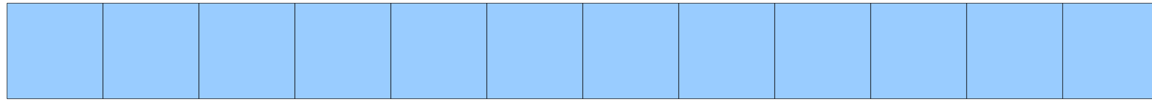
Suppose we get **unlucky**

n

n - 1

n - 2

n - 3



...

$O(n^2)$

Quicksort is Strange

- In most cases, quicksort is O(n log n).
- However, in the worst case, quicksort is $O(n^2)$.
- How can you avoid this?
- **Pick better pivots!**
  - Pick the median.
    - Can be done in O(n), but *expensive* O(n).
  - Pick the “median-of-three.”
    - Better than nothing, but still can hit worst case.
  - Pick randomly.
    - Extremely low probability of $O(n^2)$.

Quicksort is Fast

- Although quicksort is $O(n^2)$ in the worst case, it is one of the fastest known sorting algorithms.
- $O(n^2)$ behavior is extremely unlikely with random pivots; runtime is usually a very good $O(n \log n)$.
- It's hard to argue with the numbers…

Timing Quicksort

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

Timing Quicksort

Size	Selection Sort	Insertion Sort	“Split Sort”	Mergesort	Quicksort
10000	0.304	0.160	0.161	0.006	0.001
20000	1.218	0.630	0.387	0.010	0.002
30000	2.790	1.427	0.726	0.017	0.004
40000	4.646	2.520	1.285	0.021	0.005
50000	7.395	4.181	2.719	0.028	0.006
60000	10.584	5.635	2.897	0.035	0.008
70000	14.149	8.143	3.939	0.041	0.009
80000	18.674	10.333	5.079	0.042	0.009
90000	23.165	12.832	6.375	0.048	0.012

An Interesting Observation

- Big-O notation talks about long-term growth, but says nothing about small inputs.
- For small inputs, insertion sort can be better than mergesort or quicksort.
- Why?
  - Mergesort and quicksort both have high overhead per call.
  - Insertion sort is extremely simple.

Hybrid Sorts

- Combine multiple sorting algorithms together to get advantages of each.
- Insertion sort is good on small inputs.
- Merge sort is good on large inputs.
- Can we combine them?

Hybrid Mergesort

Hybrid Mergesort

```
void HybridMergesort(Vector<int>& v) {
   if (v.size() <= 8) {
      InsertionSort(v);
   } else {
       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];

       HybridMergesort(left);
       HybridMergesort(right);

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

Hybrid Mergesort

```
void HybridMergesort(Vector<int>& v) {
   if (v.size() <= 8) {
      InsertionSort(v);
   } else {
       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];

       HybridMergesort(left);
       HybridMergesort(right);

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

Runtime for Hybrid Mergesort

Size	Mergesort	Hybrid Mergesort	Quicksort
100000	0.063	0.019	0.012
300000	0.176	0.061	0.060
500000	0.283	0.091	0.063
700000	0.396	0.130	0.089
900000	0.510	0.165	0.118
1100000	0.608	0.223	0.151
1300000	0.703	0.246	0.179
1500000	0.844	0.28	0.215
1700000	0.995	0.326	0.243
1900000	1.070	0.355	0.274

Hybrid Sorts in Practice

- Introspective Sort (*Introsort)*
  - Based on quicksort, insertion sort, and *heapsort*.
  - Heapsort is $\Theta(n \log n)$ and a bit faster than mergesort.
  - Uses quicksort, then switches to heapsort if it looks like the algorithm is degenerating to $O(n^2)$.
  - Uses insertion sort for small inputs.
  - Gains the raw speed of quicksort without any of the drawbacks.

Runtime for Introsort

Size	Mergesort	Hybrid Mergesort	Quicksort
100000	0.063	0.019	0.012
300000	0.176	0.061	0.060
500000	0.283	0.091	0.063
700000	0.396	0.130	0.089
900000	0.510	0.165	0.118
1100000	0.608	0.223	0.151
1300000	0.703	0.246	0.179
1500000	0.844	0.28	0.215
1700000	0.995	0.326	0.243
1900000	1.070	0.355	0.274

Runtime for Introsort

Size	Mergesort	Hybrid Mergesort	Quicksort	Introsort
100000	0.063	0.019	0.012	0.009
300000	0.176	0.061	0.060	0.028
500000	0.283	0.091	0.063	0.043
700000	0.396	0.130	0.089	0.060
900000	0.510	0.165	0.118	0.078
1100000	0.608	0.223	0.151	0.092
1300000	0.703	0.246	0.179	0.107
1500000	0.844	0.28	0.215	0.123
1700000	0.995	0.326	0.243	0.139
1900000	1.070	0.355	0.274	0.158

We've spent all of our time talking about **fast** and **efficient** sorting algorithms.

However, we have neglected to find **slow** and **inefficient** sorting algorithms.

Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms

Hermann Gruber[1] and Markus Holzer[2] and Oliver Ruepp[2]

[1] Institut für Informatik, Ludwig-Maximilians-Universität München,
Oettingenstraße 67, D-80538 München, Germany
email: gruberh@tcs.ifi.lmu.de
[2] Institut für Informatik, Technische Universität München,
Boltzmannstraße 3, D-85748 Garching bei München, Germany
email: {holzer,ruepp}@in.tum.de

Introducing Bogosort

- Intuition:
  - Suppose you want to sort a deck of cards.
  - Check if it's sorted.
  - If so, you're done.
  - Otherwise, shuffle the deck and repeat.
- This is $O(n \times n!)$ in the average case.
- Could run for **much** longer...

Further Topics in Sorting

- Heapsort
  - Extremely fast $\Theta(n \log n)$ sorting algorithm.
- Radix Sort
  - Sorts integers in $O(n \log U)$ by working one digit at a time, where U is the largest integer being sorted.
  - Used by old IBM sorting machines.
- Bead Sort
  - Ingenious sorting algorithm for integers.
  - Uses gravity... how cool is that?