Algorithmic Analysis and Sorting, Part One

Announcements

- YEAH hours for Assignment 3 today, 4:15 – 5:30PM in 370-370.
- Solutions to warm-up problems will be posted later today.
- Alternate midterms – please email Zach soon if you'd like to take the exam on a different date.

Fundamental Question:

How can we compare solutions to problems?

One Idea: **Runtime**

Why Runtime Isn't a Good Metric

- Fluctuates between computer to computer and from run to run.
- Fluctuates based on inputs.
- Doesn't predict behavior for larger inputs.

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

  return false;
}
```

Work Done: At most $k_0 n + k_1$

Big Observations

- Don't need to explicitly compute these constants.
  - Whether runtime is $4n + 10$ or $100n + 137$, runtime is still proportional to input size.
  - Can just plot the runtime to obtain actual values.
- Only the dominant term matters.
  - For both $4n + 1000$ and $n + 137$, for very large n most of the runtime is explained by n.
- Is there a concise way of describing this?

Big-O

Big-O Notation

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

Formally...

$f(n) = O(g(n))$ if there are constants n_0 and c such that for any $n > n_0$, $|f(n)| \leq c|g(n)|$

In other words, big-O is an **upper bound** on a function for large inputs to that function.

Algorithmic Analysis with Big-O

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();
}
```

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();
}
```

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)

A More Interesting Example

A More Interesting Example

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

  return false;
}
```

A More Interesting Example

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

  return false;
}
```

How do we analyze this?

Types of Analysis

- Worst-Case Analysis
  - What's the *worst* possible runtime for the algorithm?
  - Useful for "sleeping well at night."
- Best-Case Analysis
  - What's the *best* possible runtime for the algorithm?
  - Useful to see if the algorithm performs well in some cases.
- Average-Case Analysis
  - What's the *average* runtime for the algorithm?
  - Far beyond the scope of this class.

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.

A More Interesting Example

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

  return false;
}
```

O(n)

Determining if a Character is a Letter

Determining if a Character is a Letter

```
bool IsAlpha(char ch) {
    return (ch >= 'A' && ch <= 'Z') ||
           (ch >= 'a' && ch <= 'z');
}
```

Determining if a Character is a Letter

```
bool IsAlpha(char ch) {
    return (ch >= 'A' && ch <= 'Z') ||
           (ch >= 'a' && ch <= 'z');
}
```

O(1)

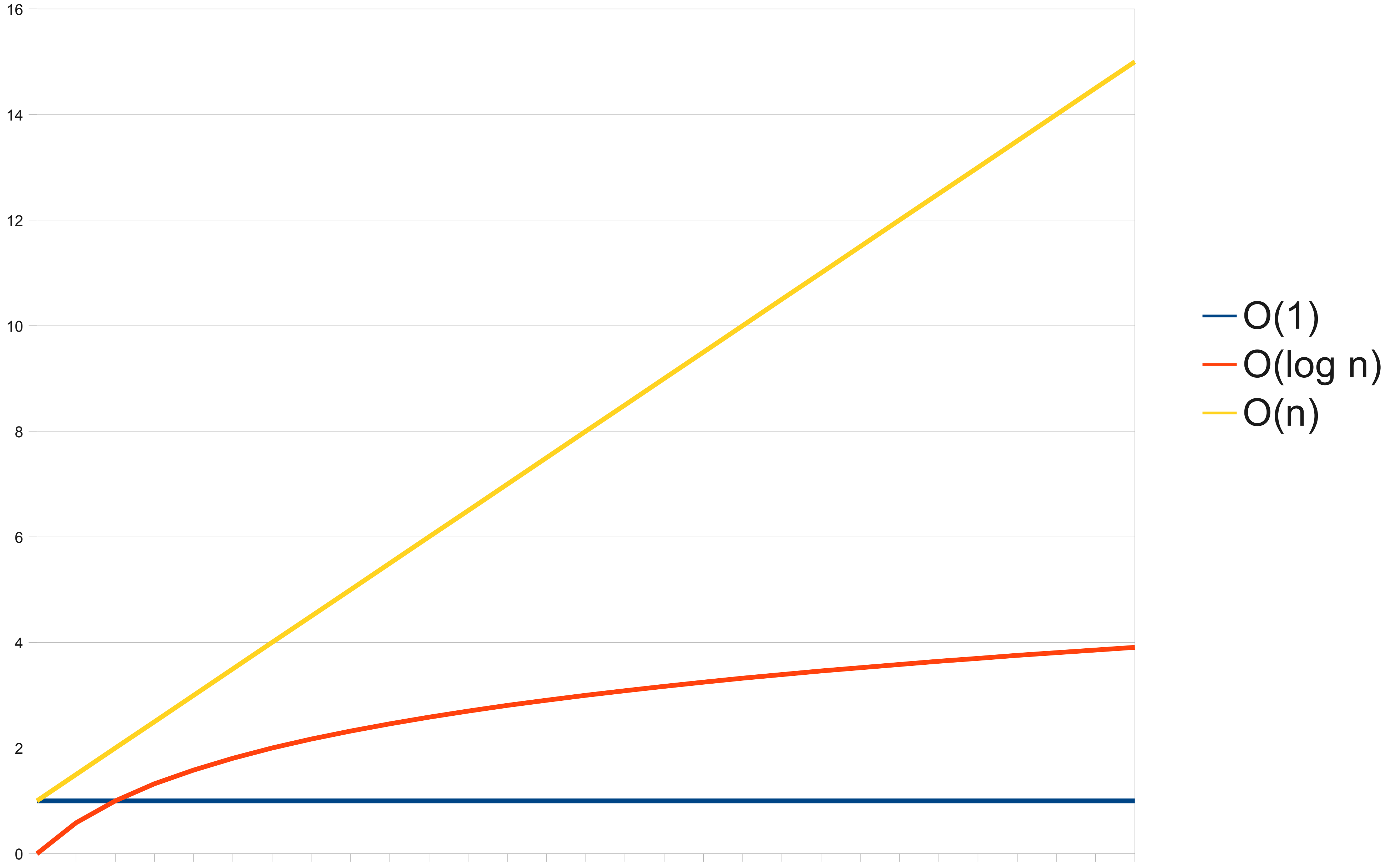
What Can Big-O Tell Us?

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

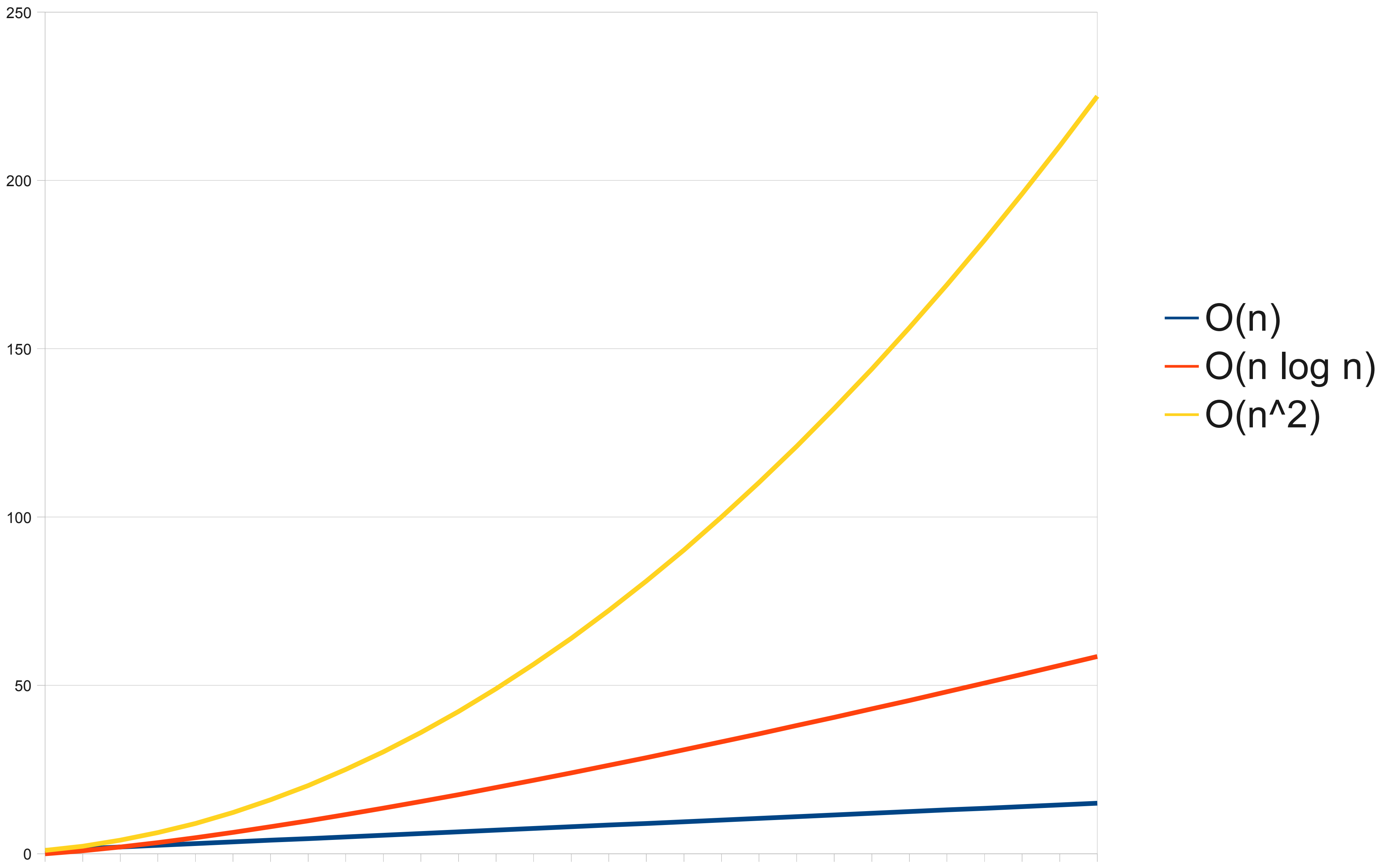
What *Can't* Big-O Tell Us?

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

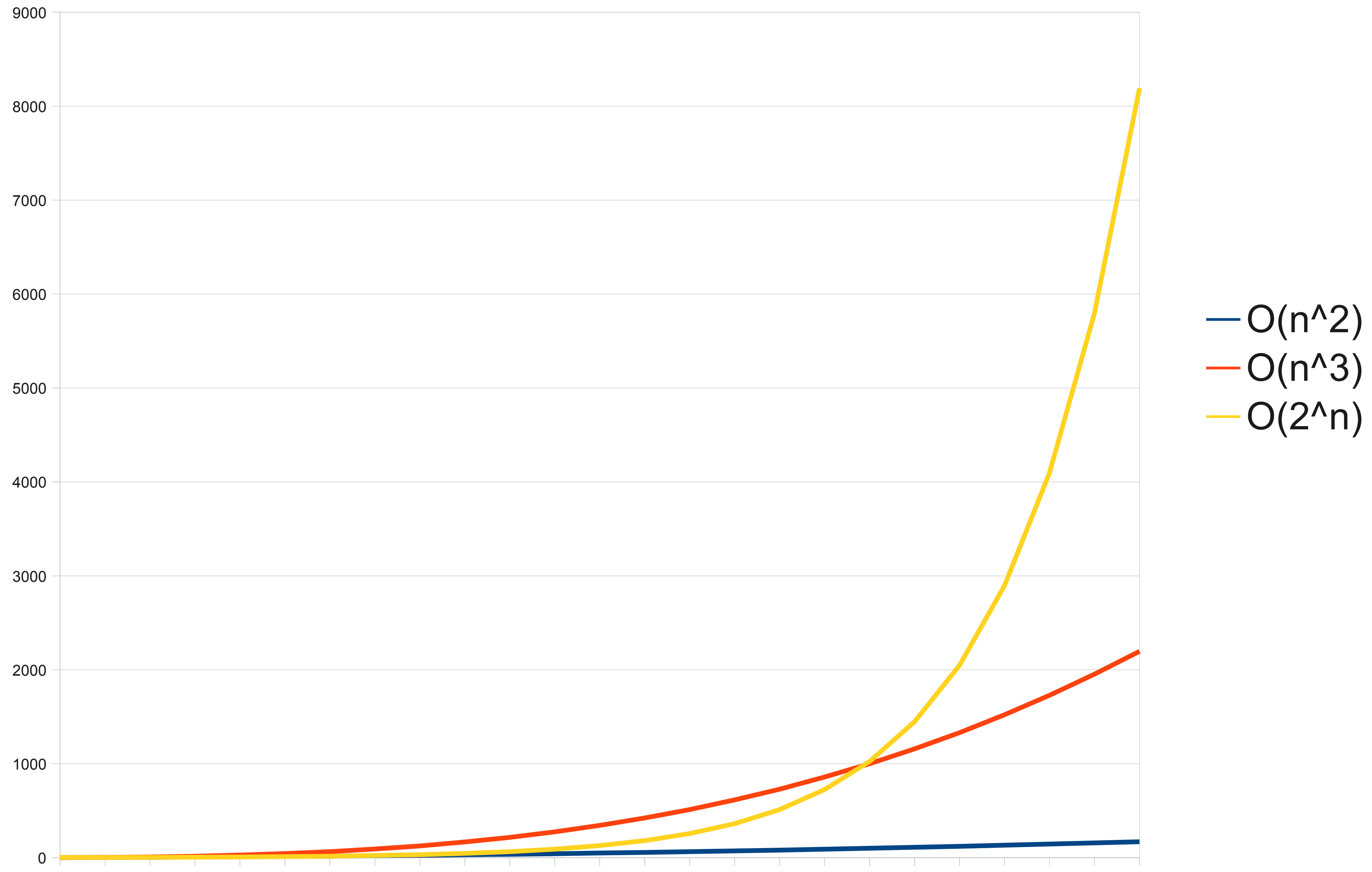
Growth Rates, Part One



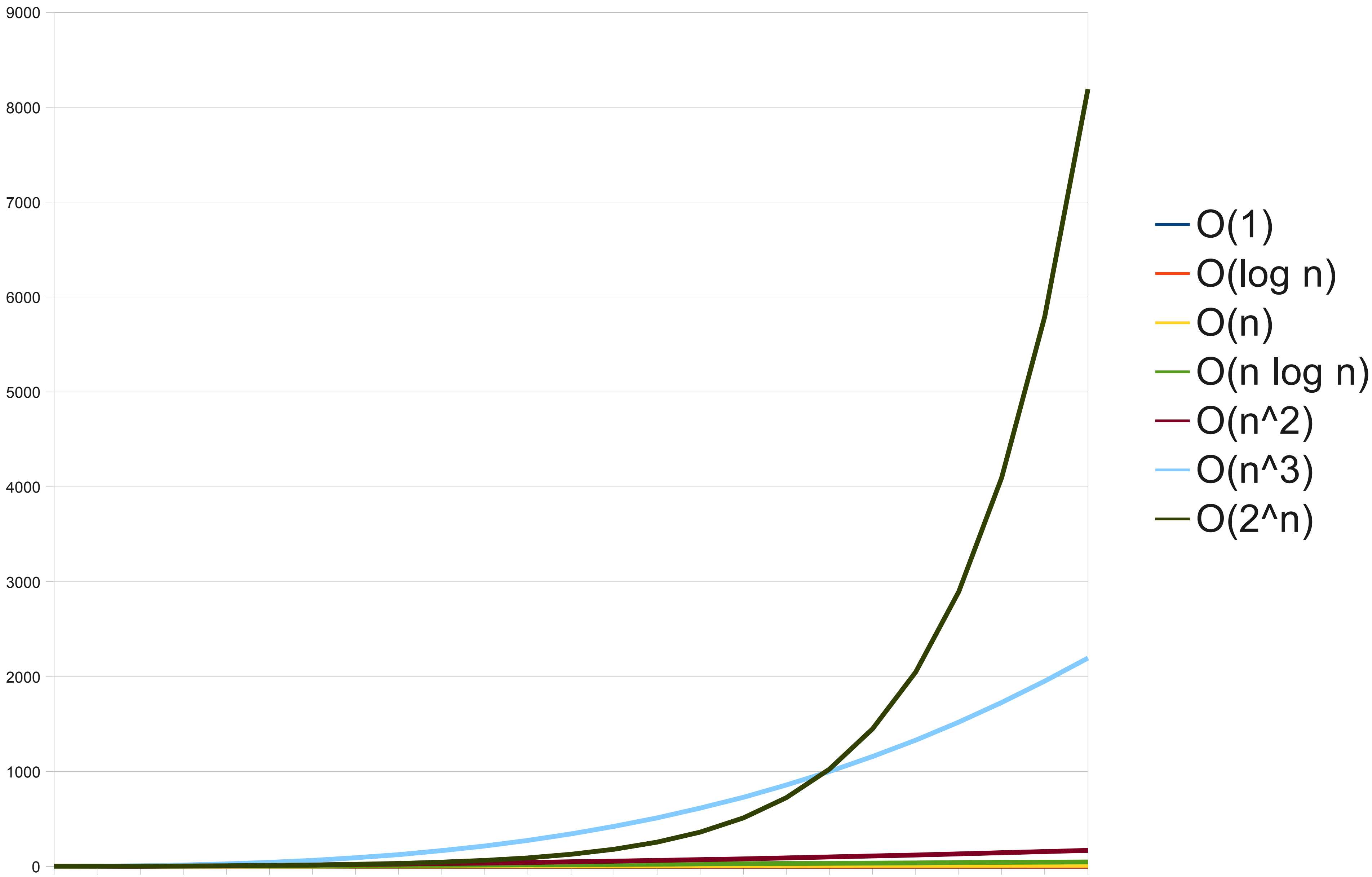
Growth Rates, Part Two



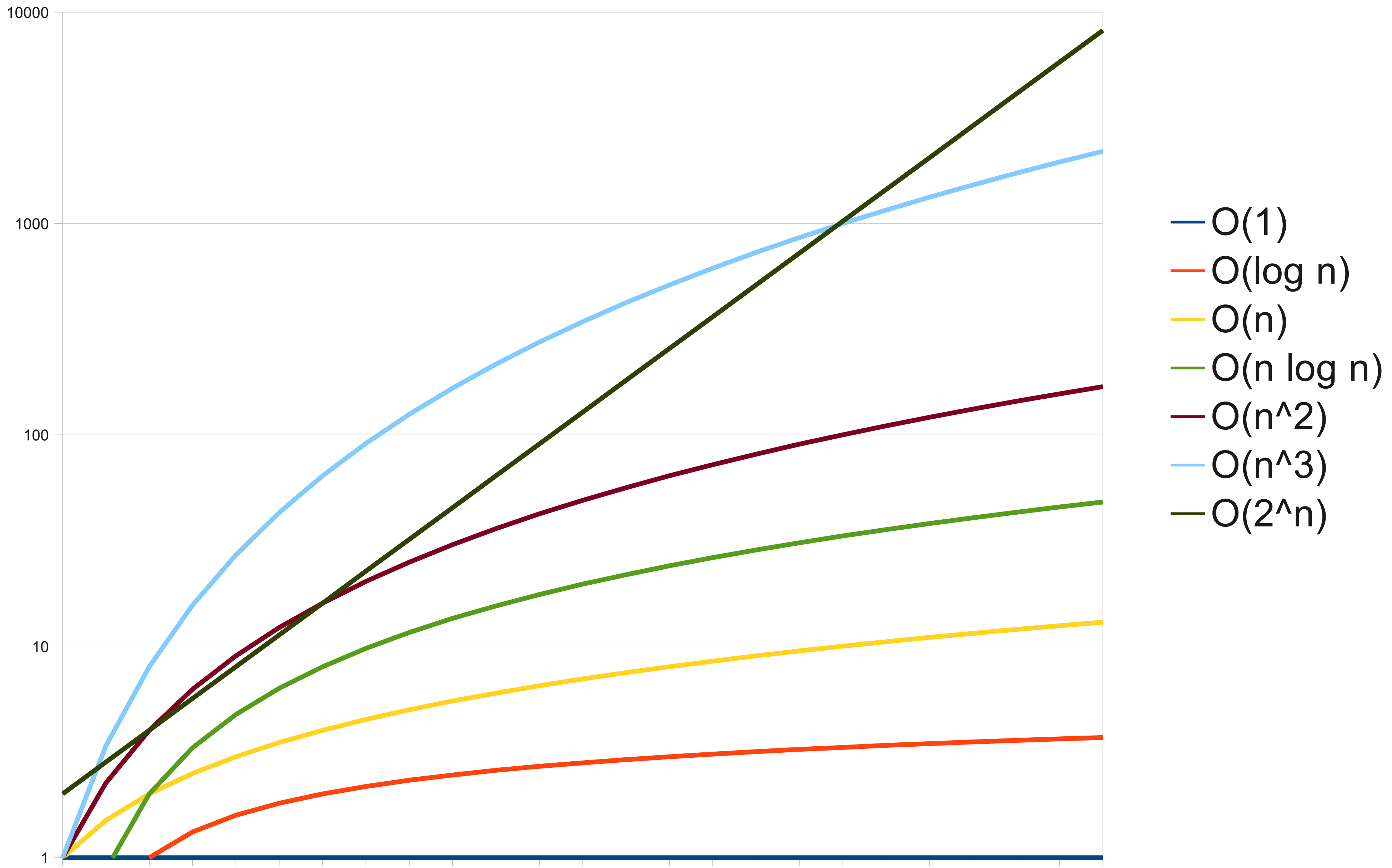
Growth Rates, Part Three



To Give You A Better Sense...



Once More with Logarithms



Comparison of Runtimes

(1 operation = 1 microsecond)

Size	1	lg n	n	n log n	n^2	n^3
100	1μs	7μs	100μs	0.7ms	10ms	<1min
200	1μs	8μs	200μs	1.5ms	40ms	<1min
300	1μs	8μs	300μs	2.5ms	90ms	1min
400	1μs	9μs	400μs	3.5ms	160ms	2min
500	1μs	9μs	500μs	4.5ms	250ms	4min
600	1μs	9μs	600μs	5.5ms	360ms	6min
700	1μs	9μs	700μs	6.6ms	490ms	9min
800	1μs	10μs	800μs	7.7ms	640ms	12min
900	1μs	10μs	900μs	8.8ms	810ms	17min
1000	1μs	10μs	1000μs	10ms	1000ms	22min
1100	1μs	10μs	1100μs	11ms	1200ms	29min
1200	1μs	10μs	1200μs	12ms	1400ms	37min
1300	1μs	10μs	1300μs	13ms	1700ms	45min
1400	1μs	10μs	1400μs	15ms	2000ms	56min

Summary of Big-O

- A means of describing the growth rate of a function.
- Ignores all but the leading term.
- Ignores constants.
- Allows for quantitative ranking of algorithms.
- Allows for quantiative reasoning about algorithms.

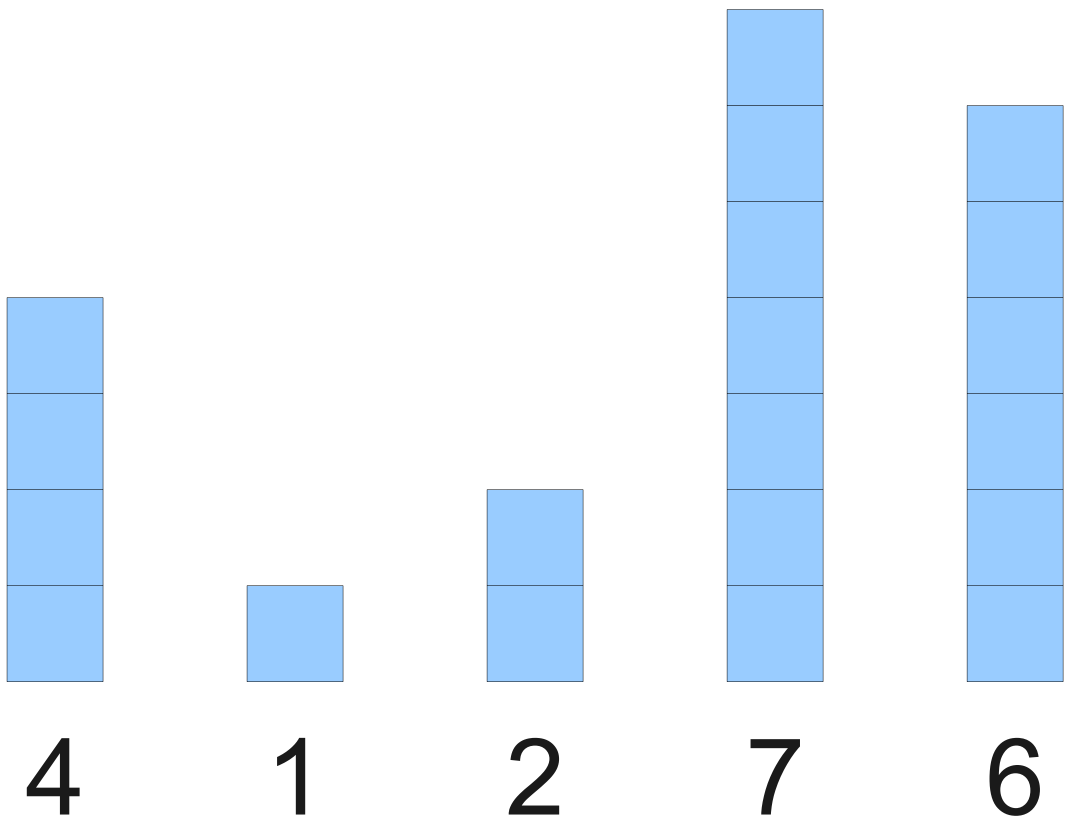
Sorting Algorithms

The Sorting Problem

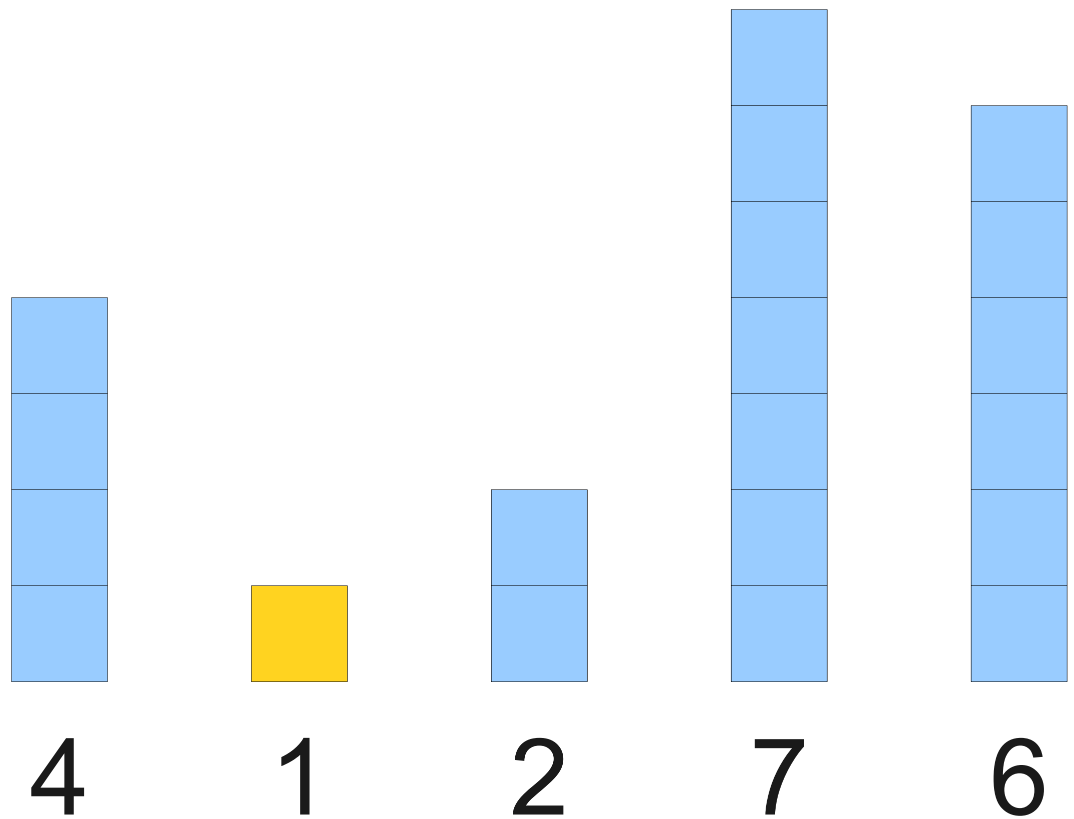
- Given a list of elements, sort those elements in ascending order.
- There are **many** ways to solve this problem.
- What is the **best** way to solve this problem?
- We'll use big-O to find out!

An Initial Idea: **Selection Sort**

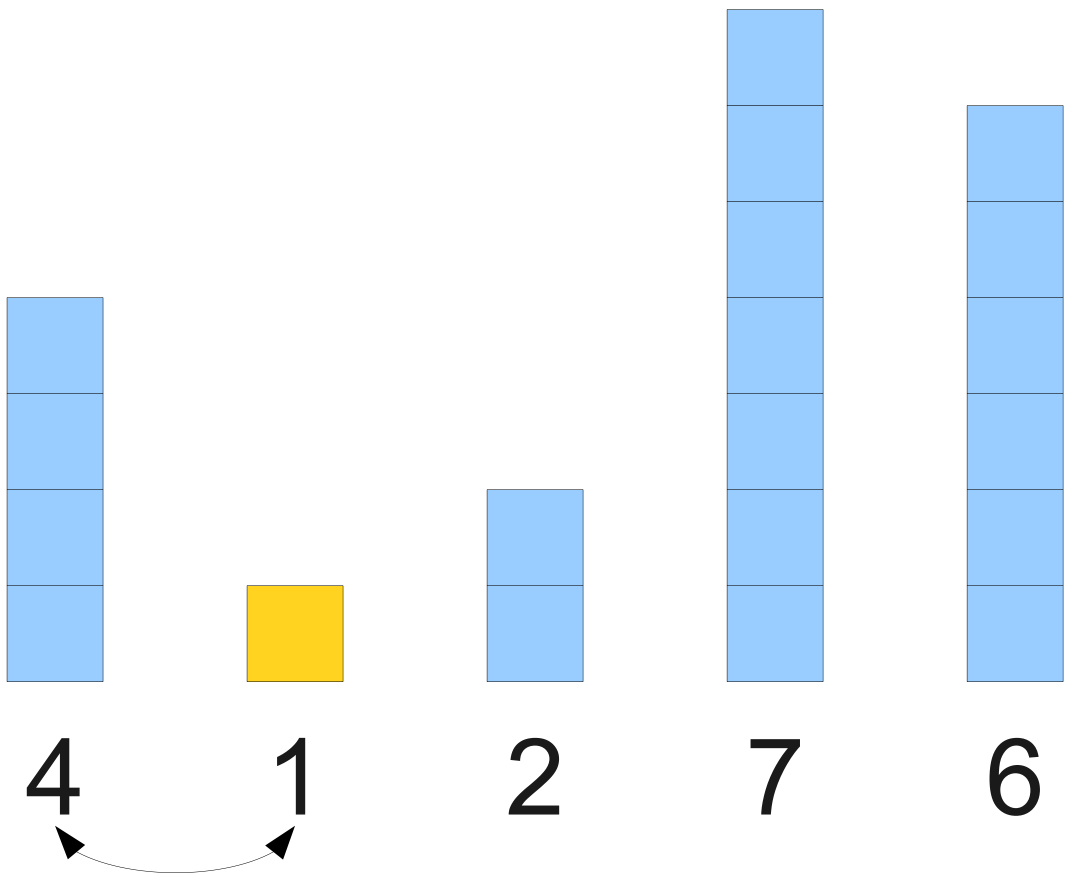
An Initial Idea: **Selection Sort**



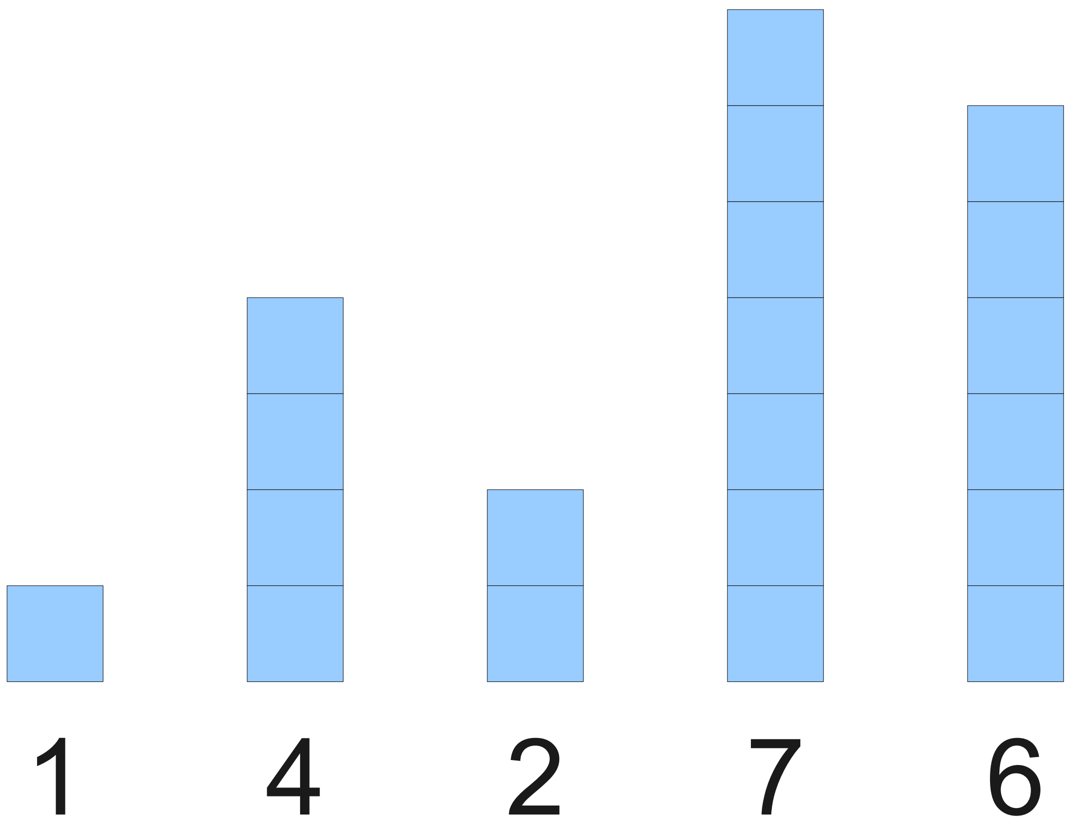
An Initial Idea: **Selection Sort**



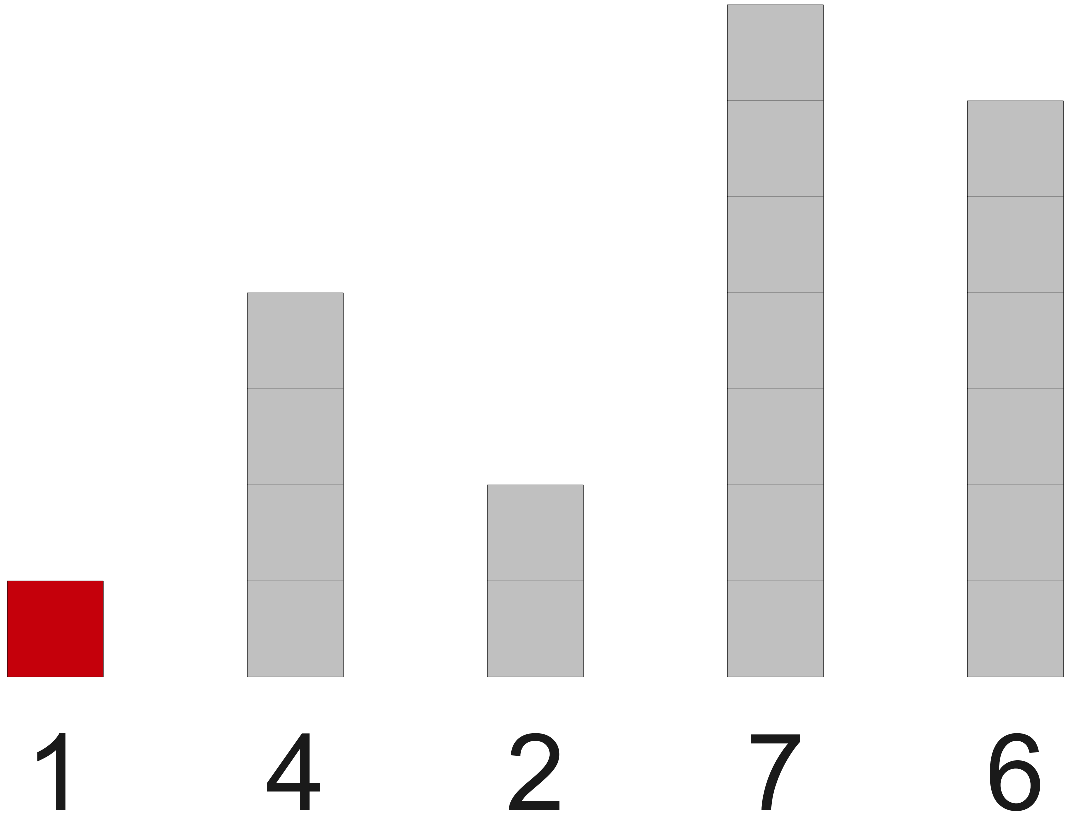
An Initial Idea: **Selection Sort**



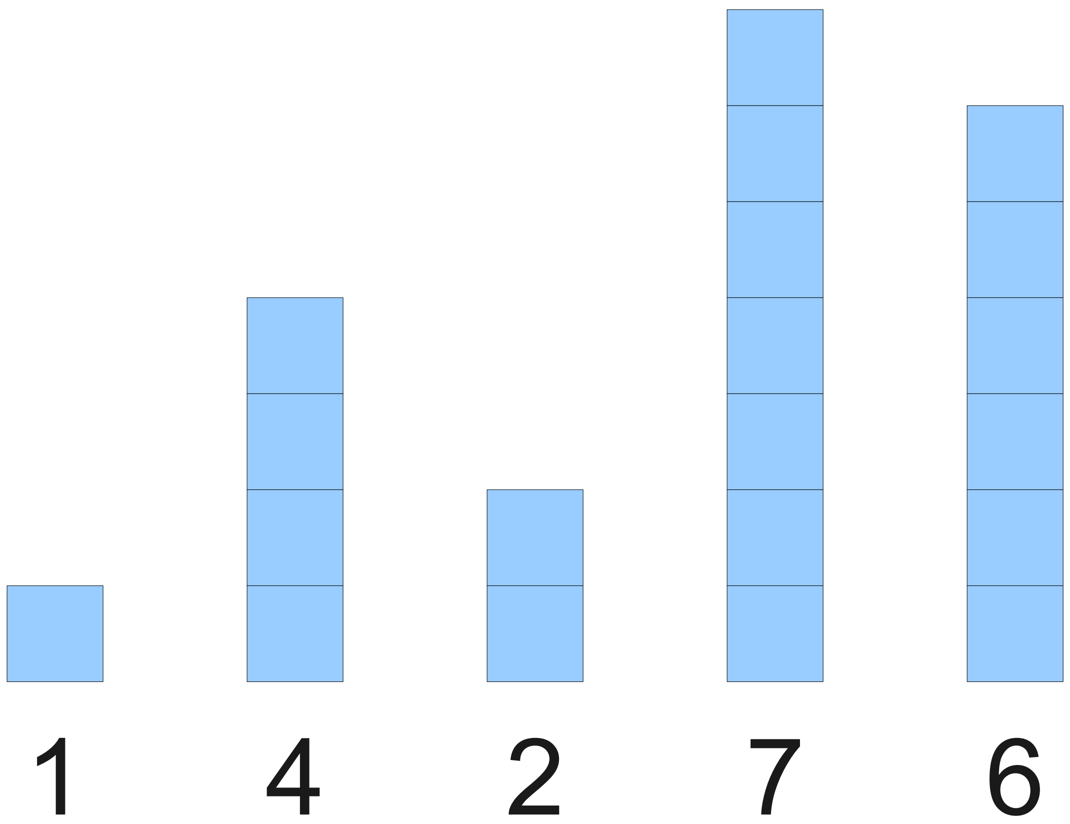
An Initial Idea: **Selection Sort**



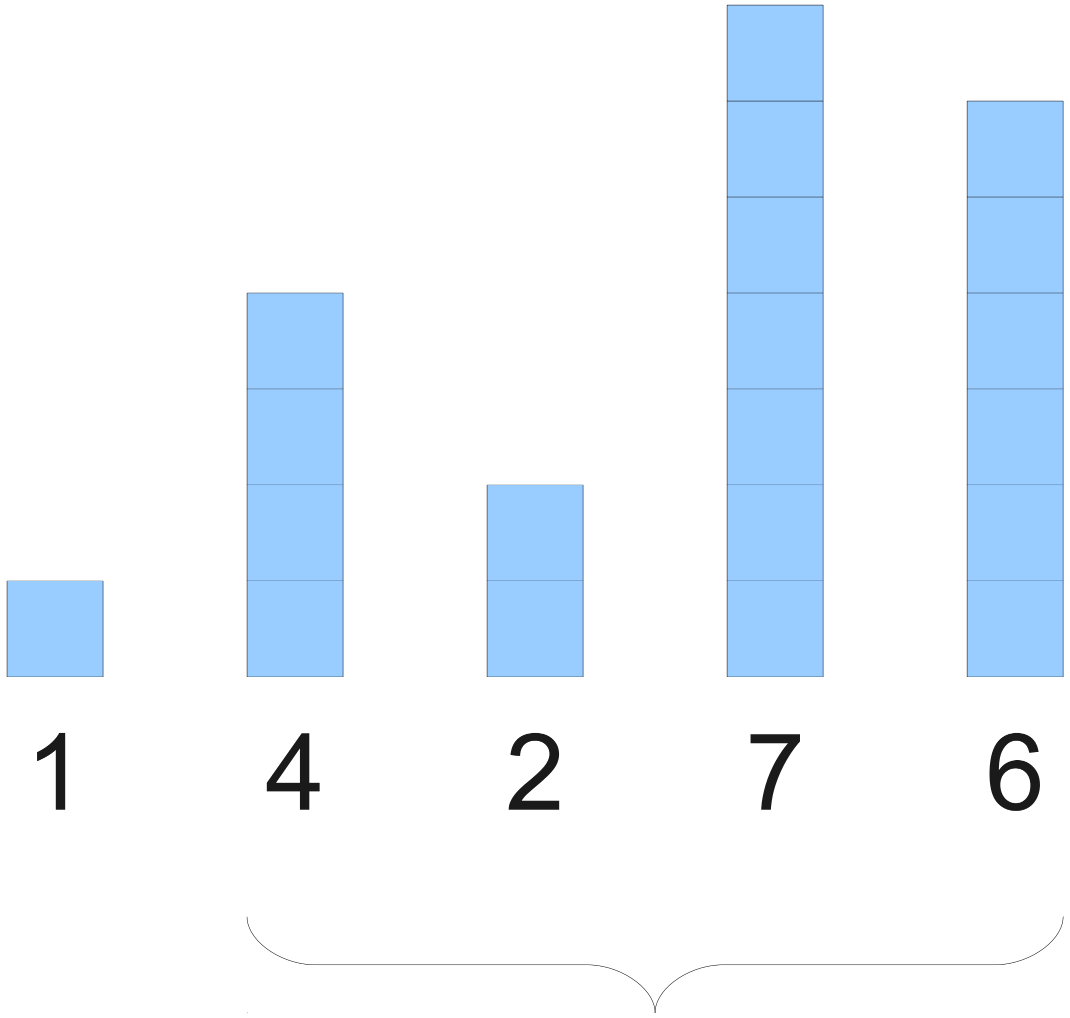
An Initial Idea: Selection Sort



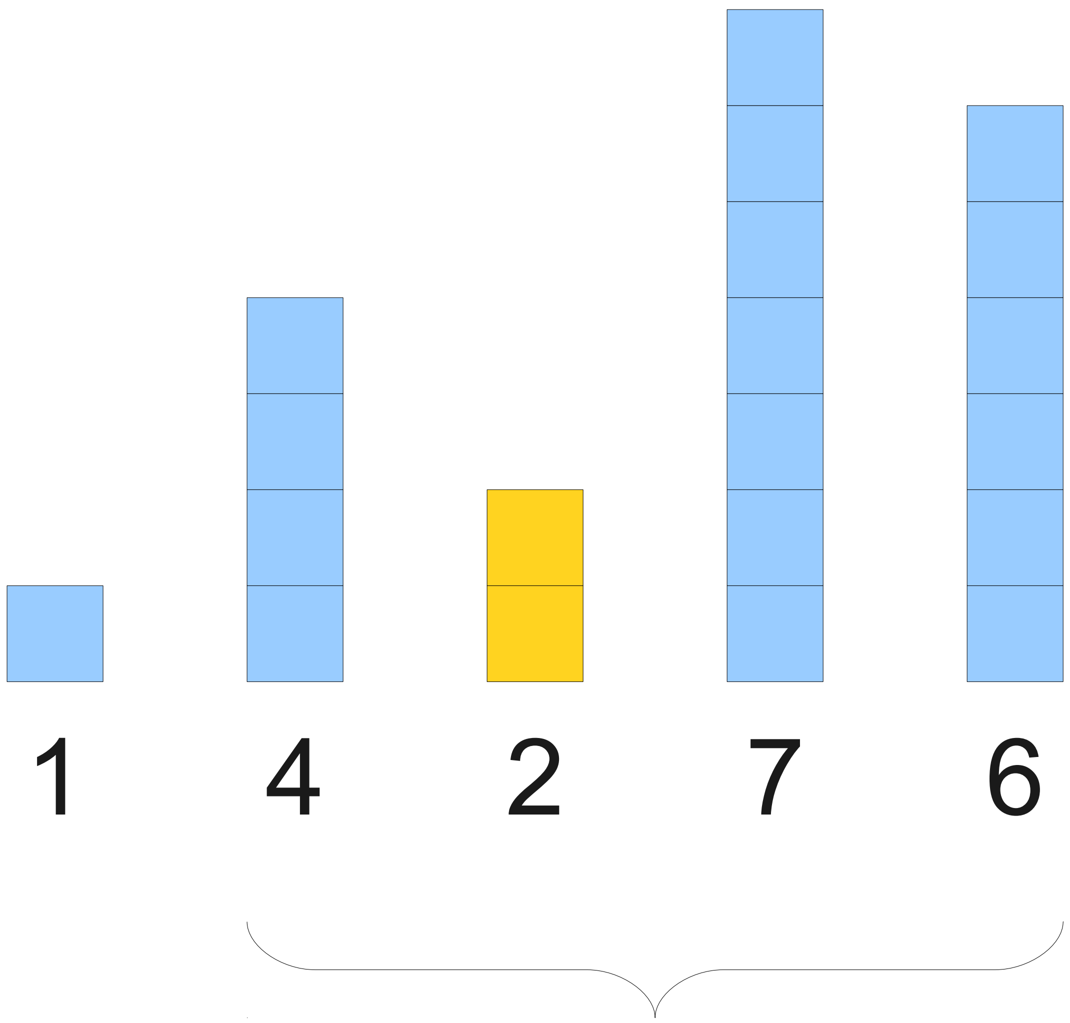
An Initial Idea: **Selection Sort**



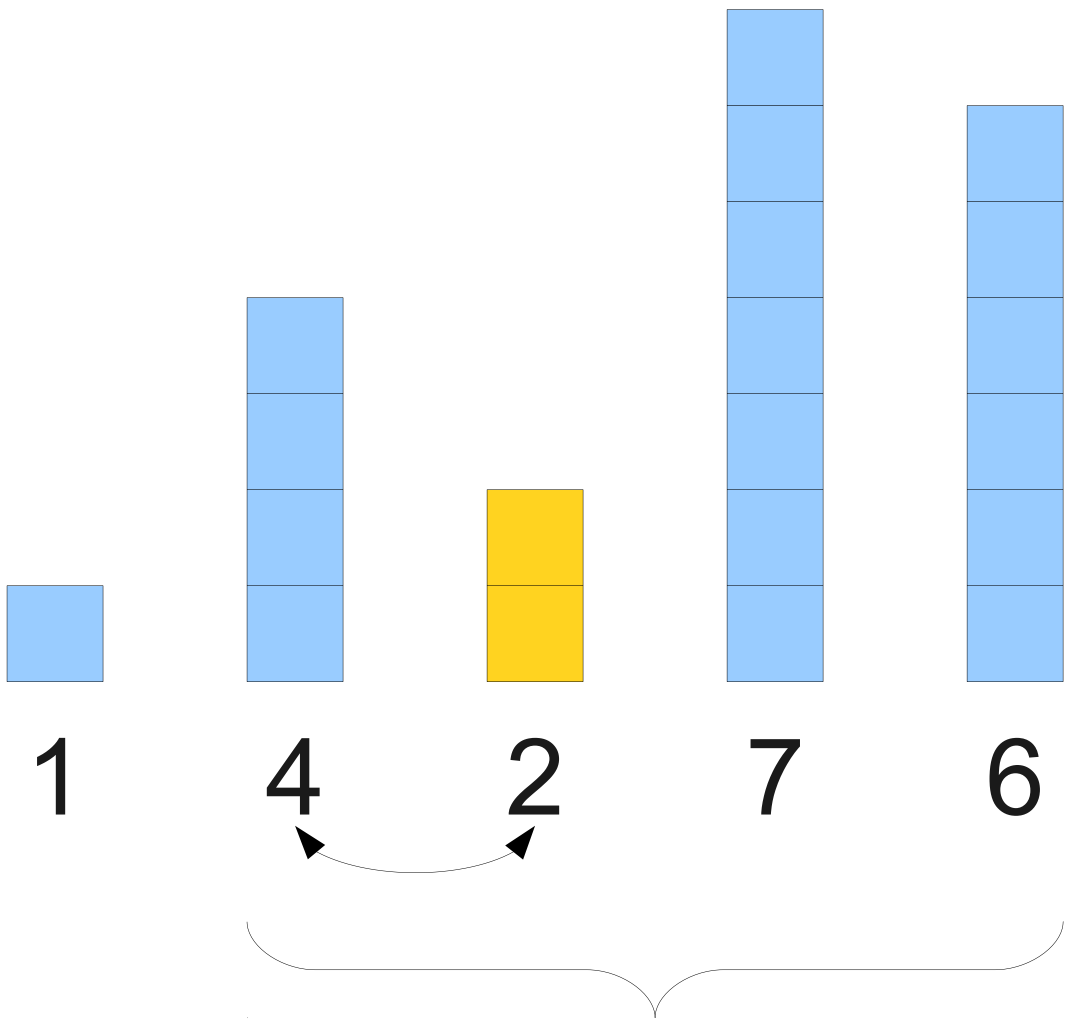
An Initial Idea: **Selection Sort**



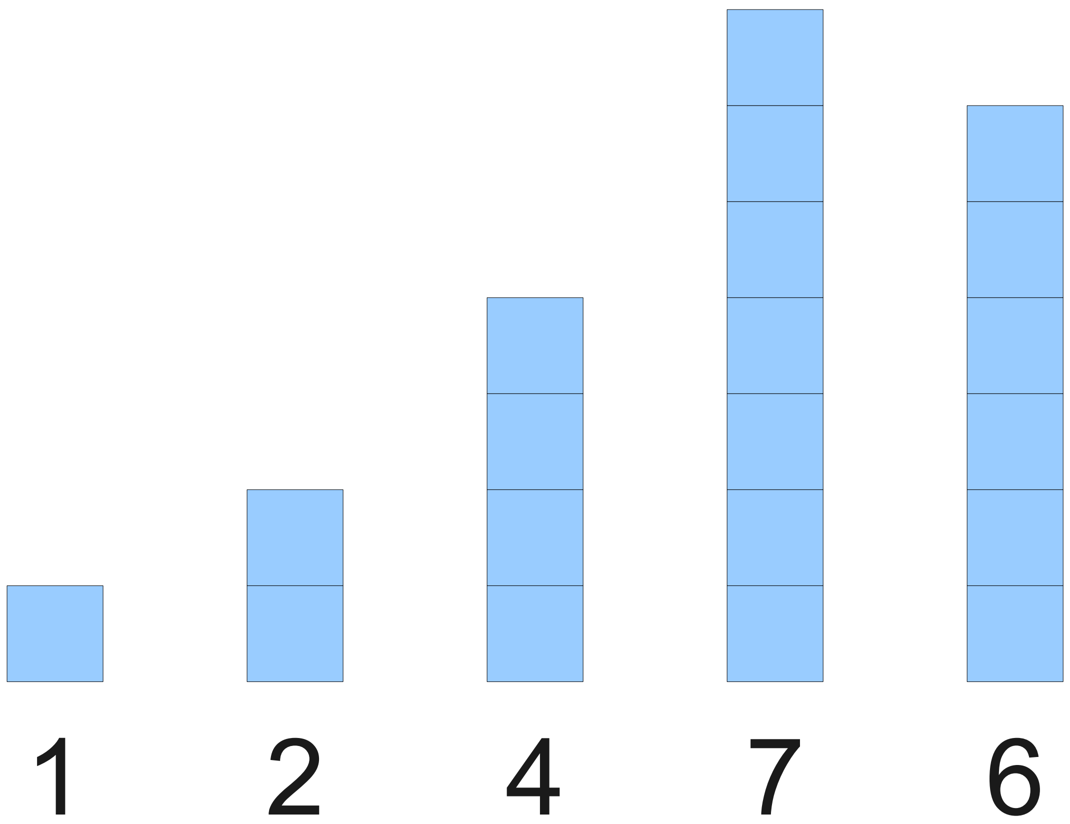
An Initial Idea: **Selection Sort**



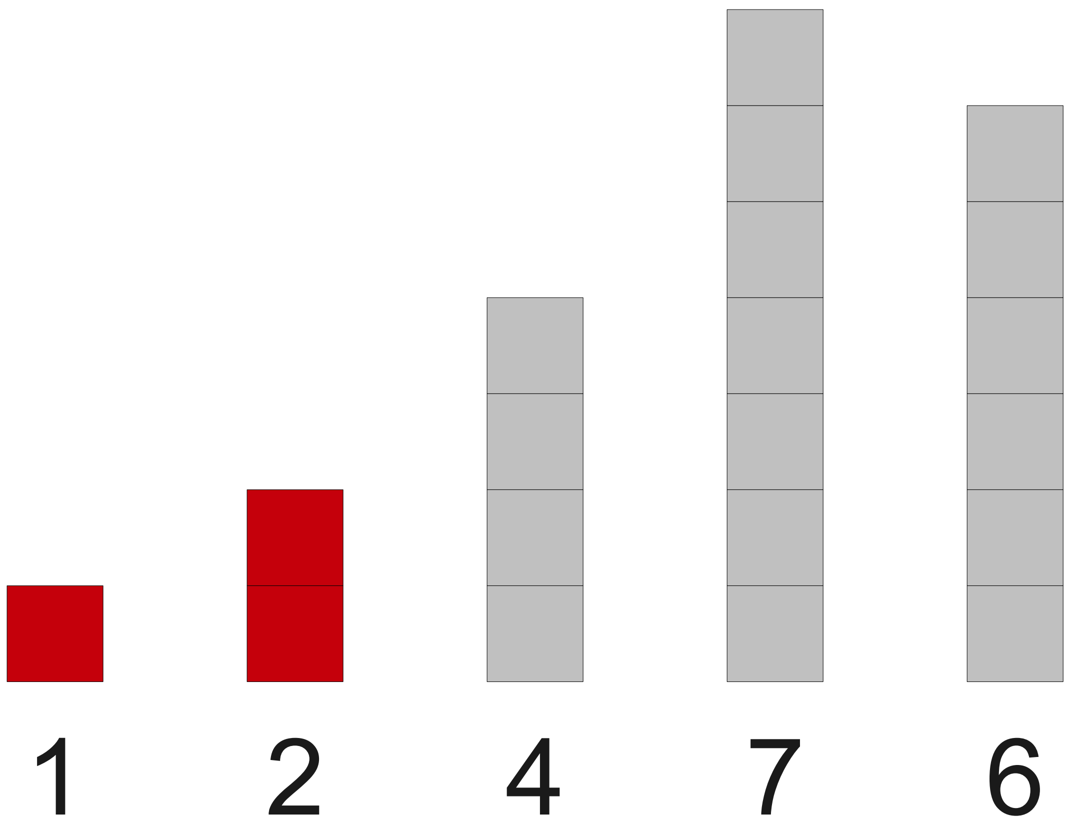
An Initial Idea: **Selection Sort**



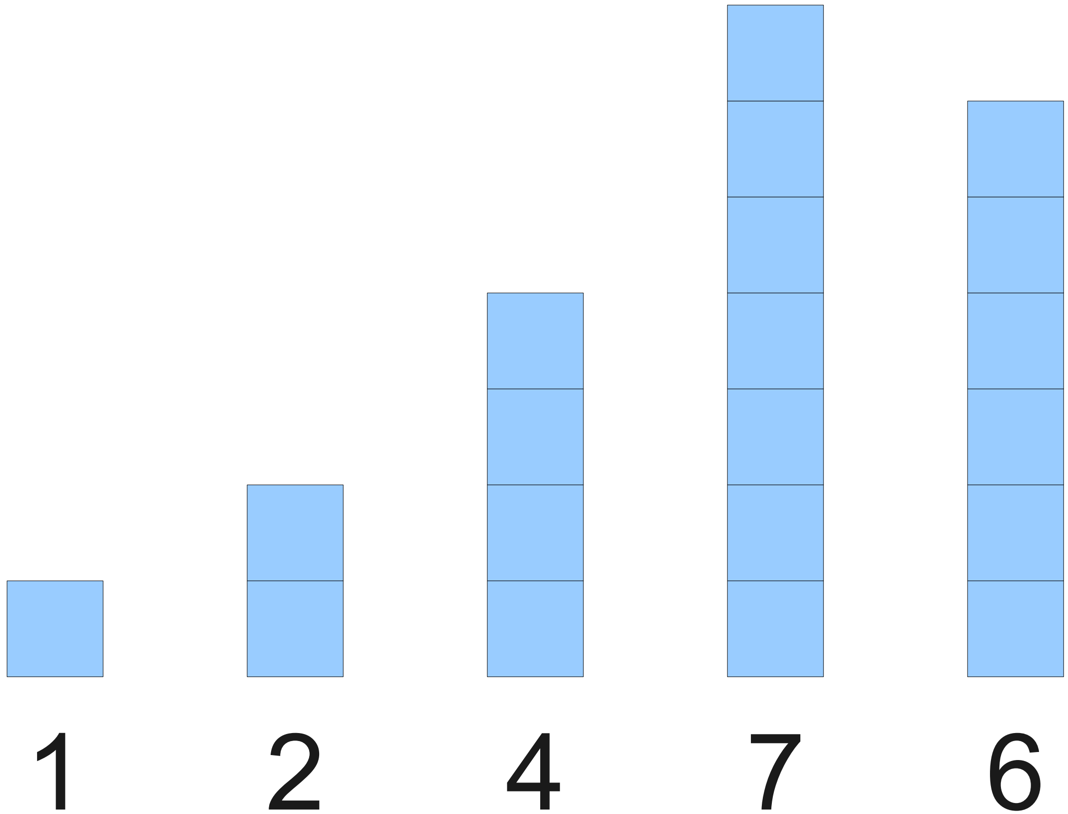
An Initial Idea: **Selection Sort**



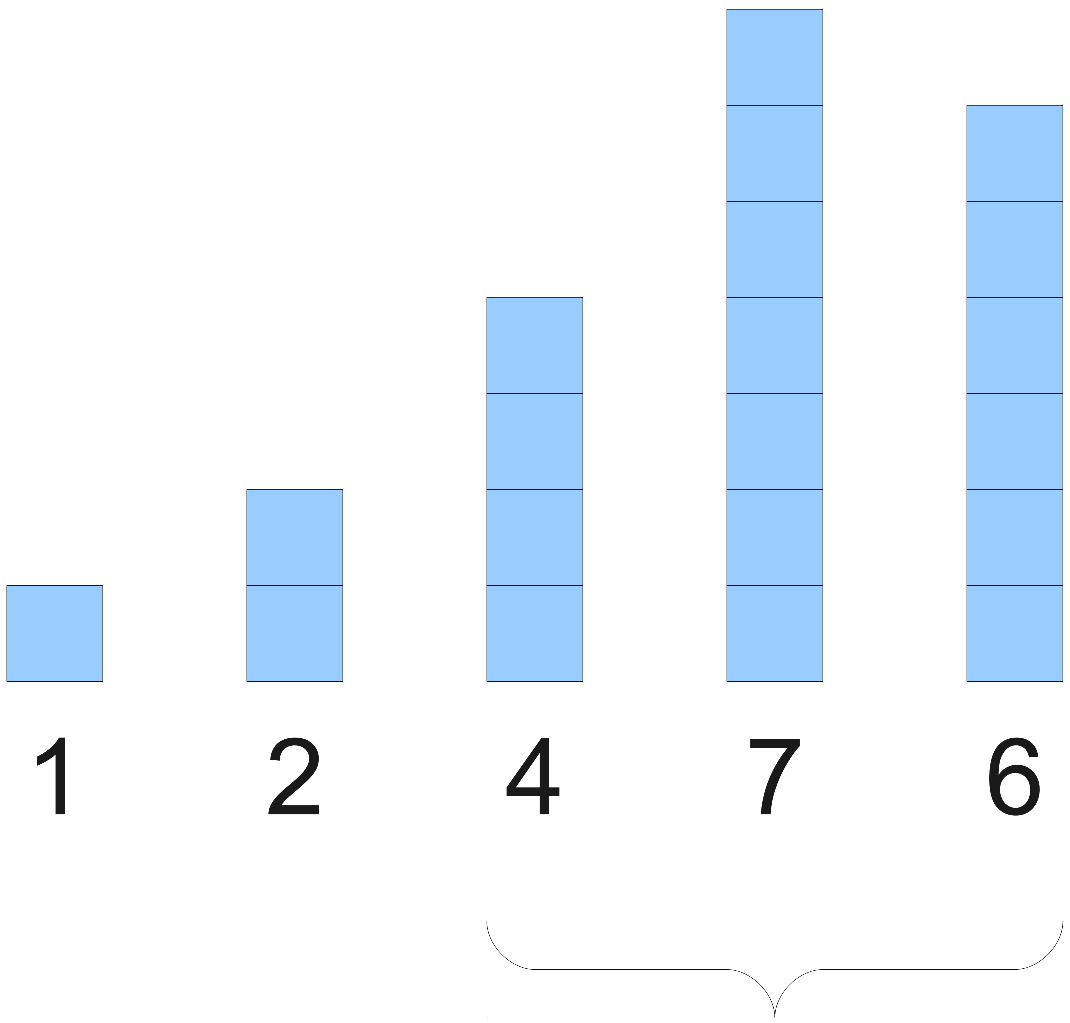
An Initial Idea: **Selection Sort**



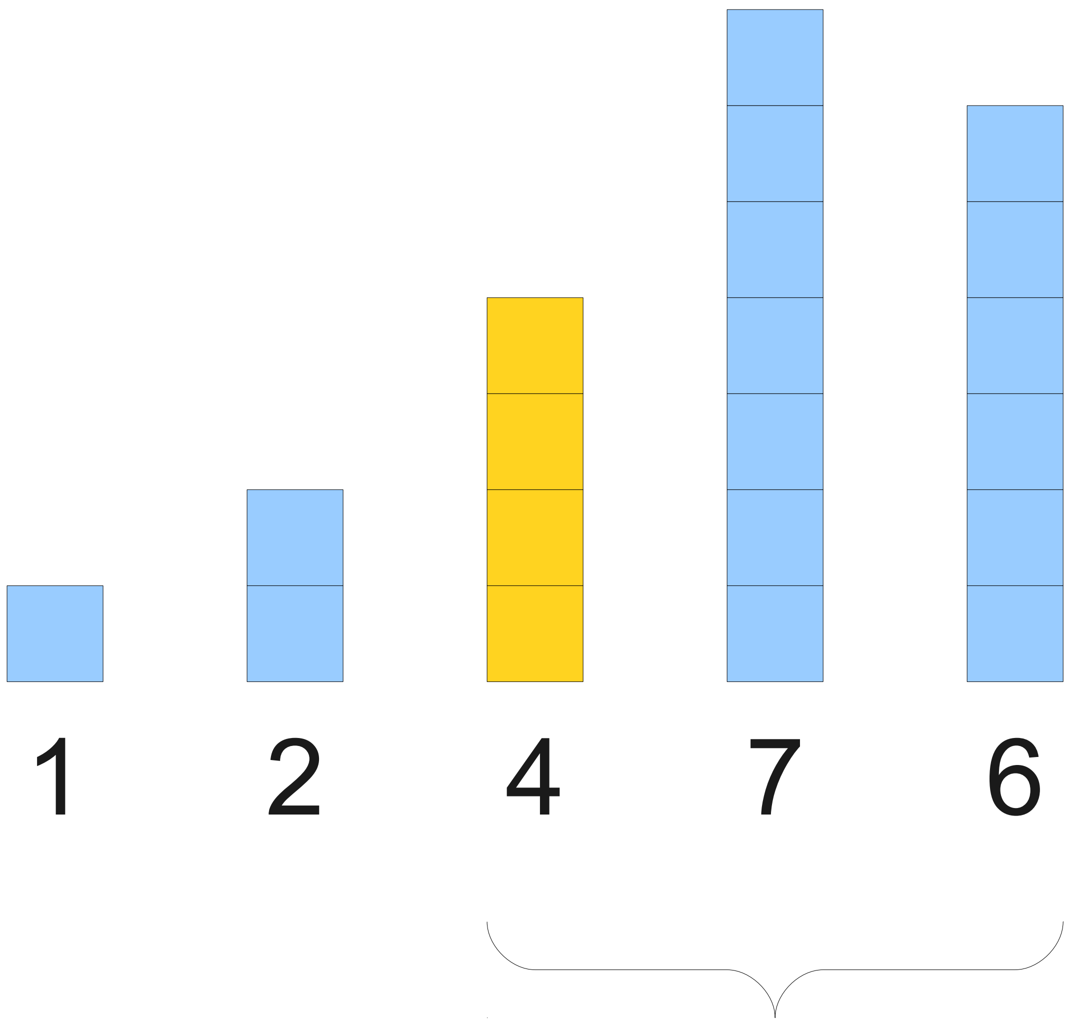
An Initial Idea: **Selection Sort**



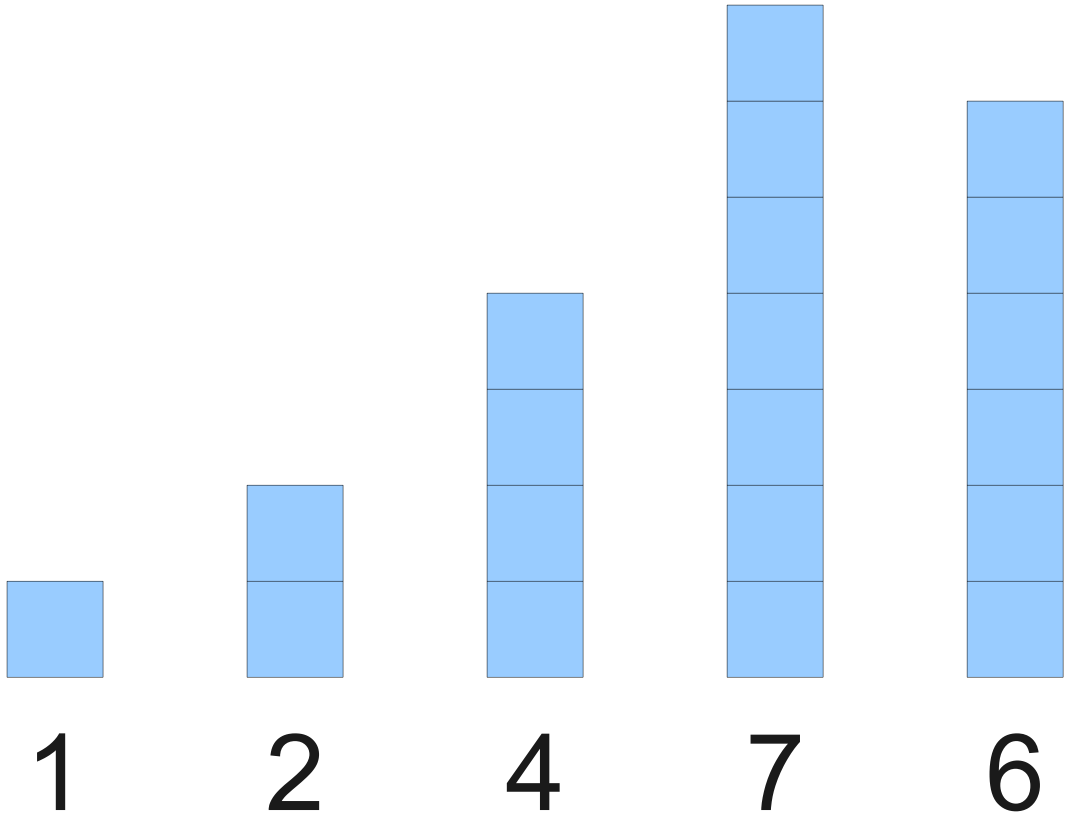
An Initial Idea: **Selection Sort**



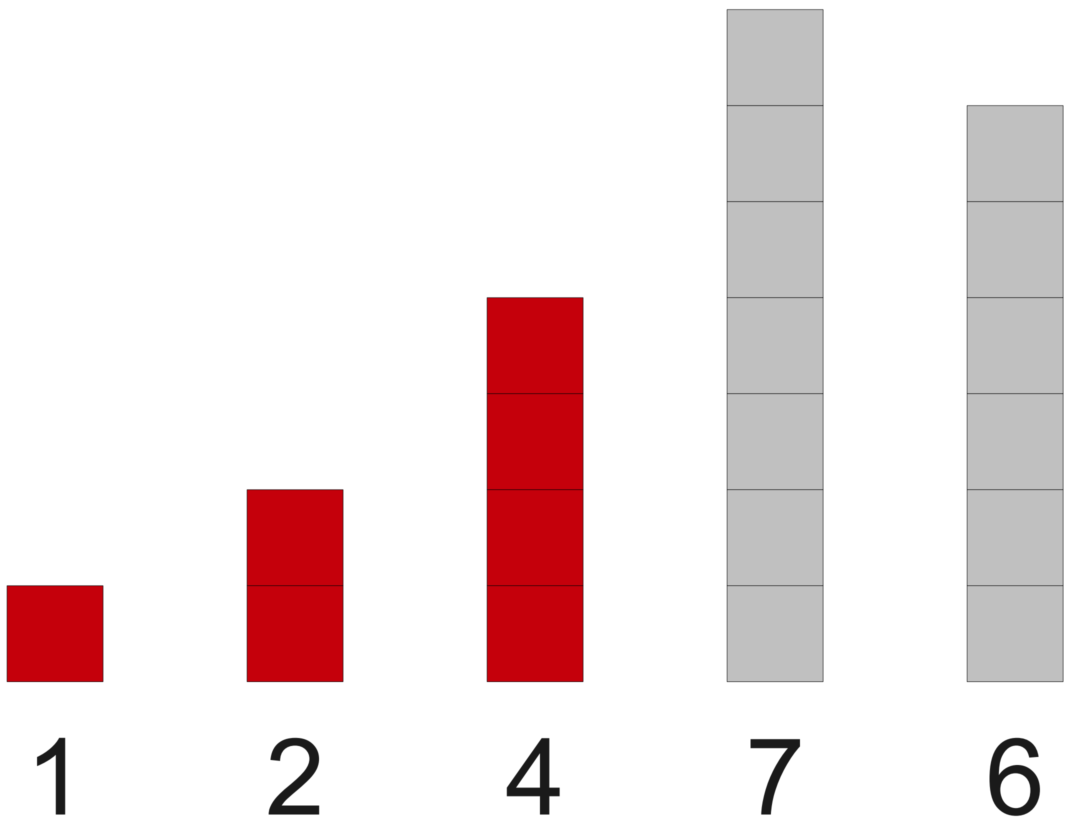
An Initial Idea: **Selection Sort**



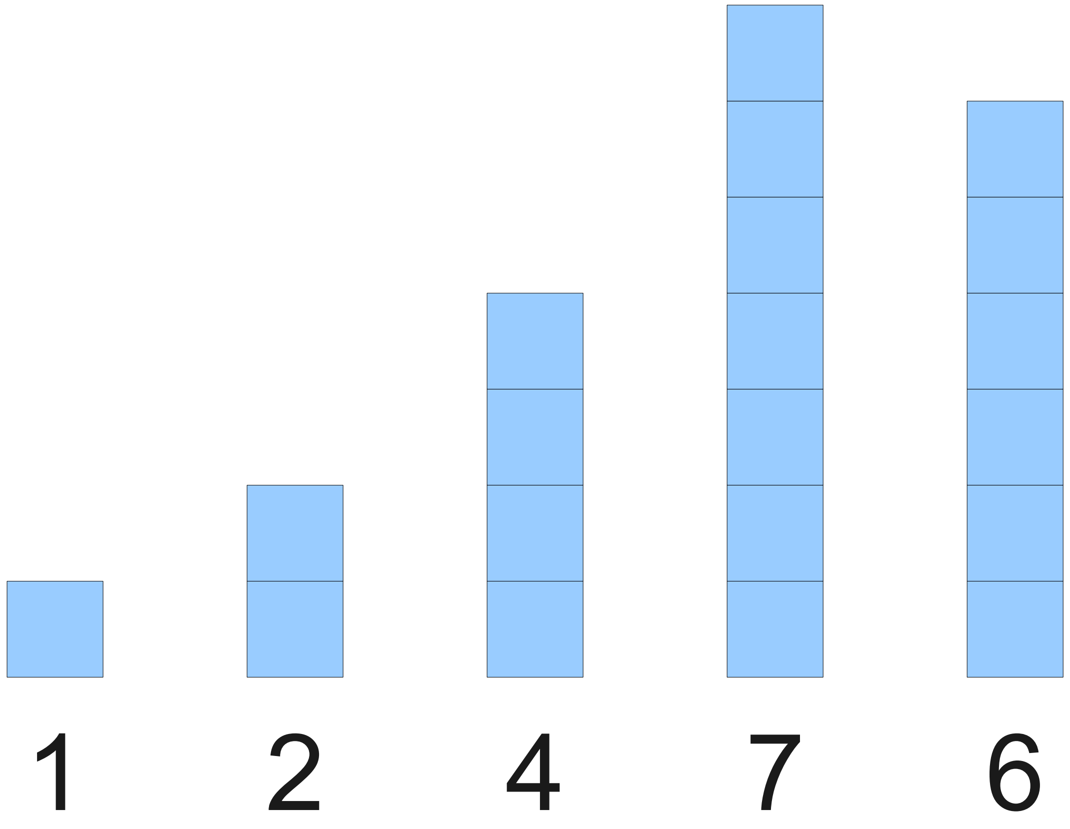
An Initial Idea: **Selection Sort**



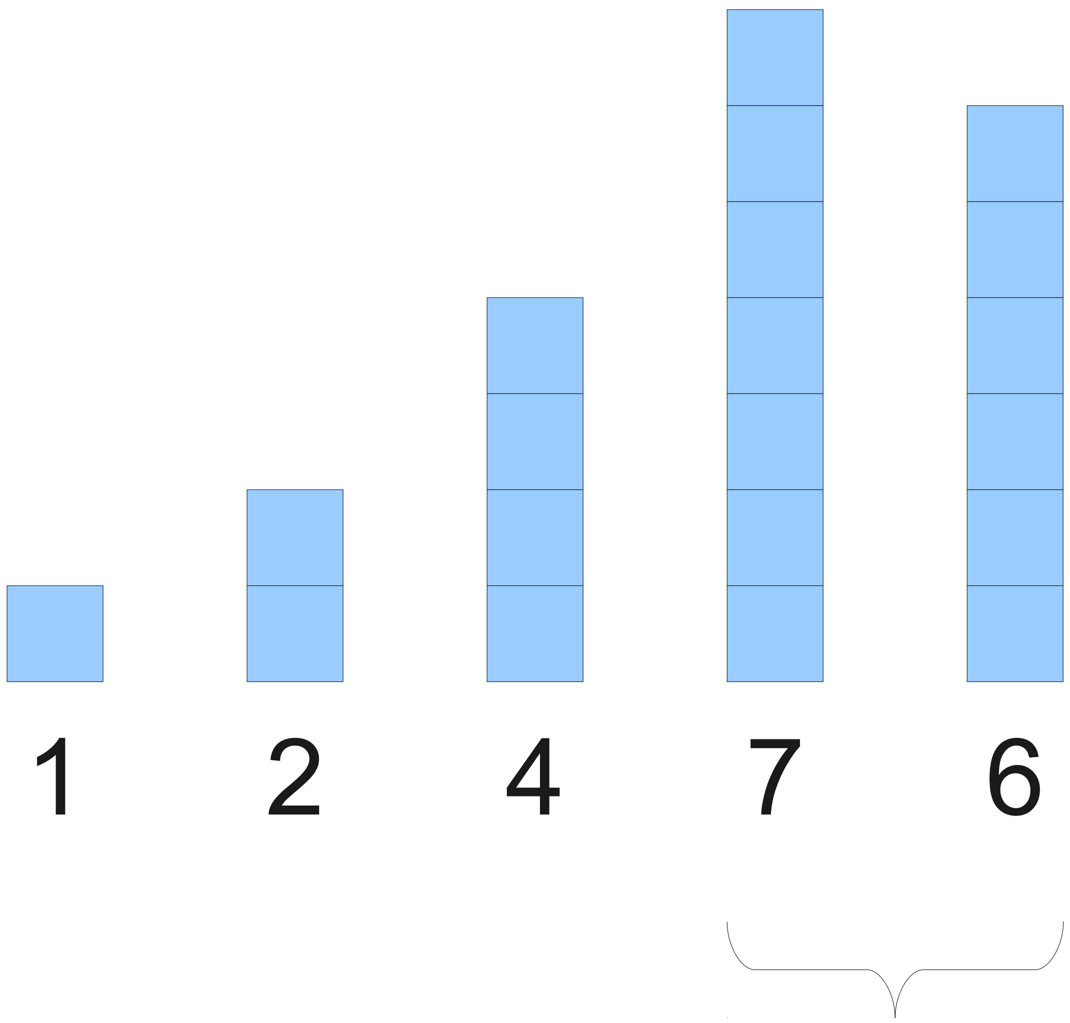
An Initial Idea: **Selection Sort**



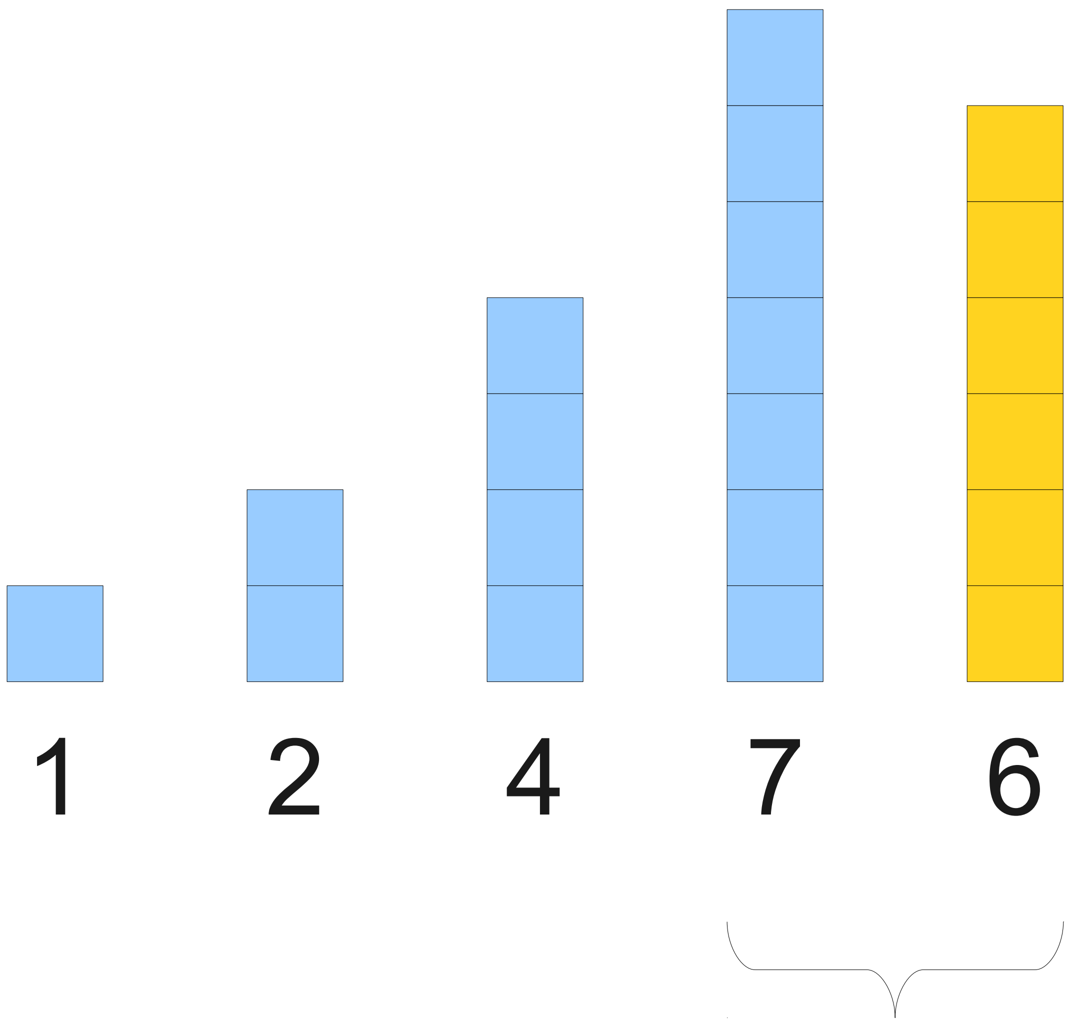
An Initial Idea: **Selection Sort**



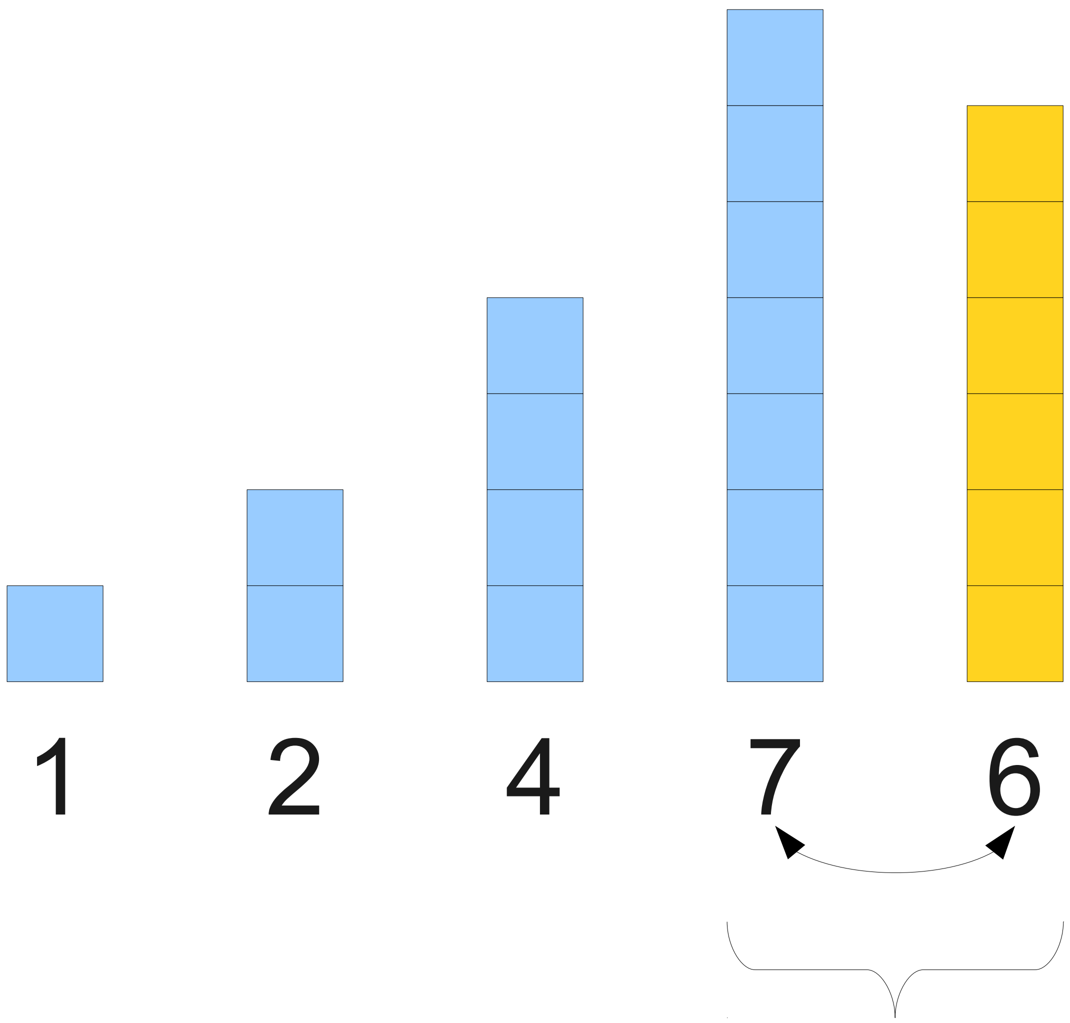
An Initial Idea: **Selection Sort**



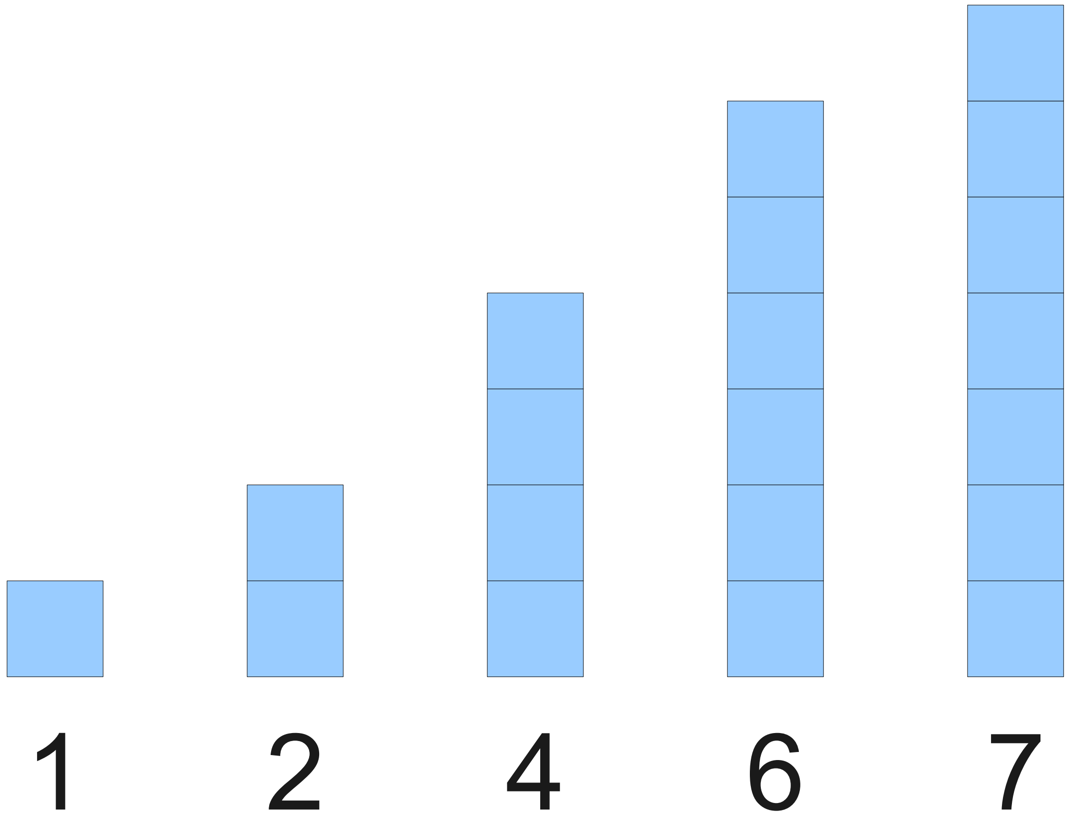
An Initial Idea: **Selection Sort**



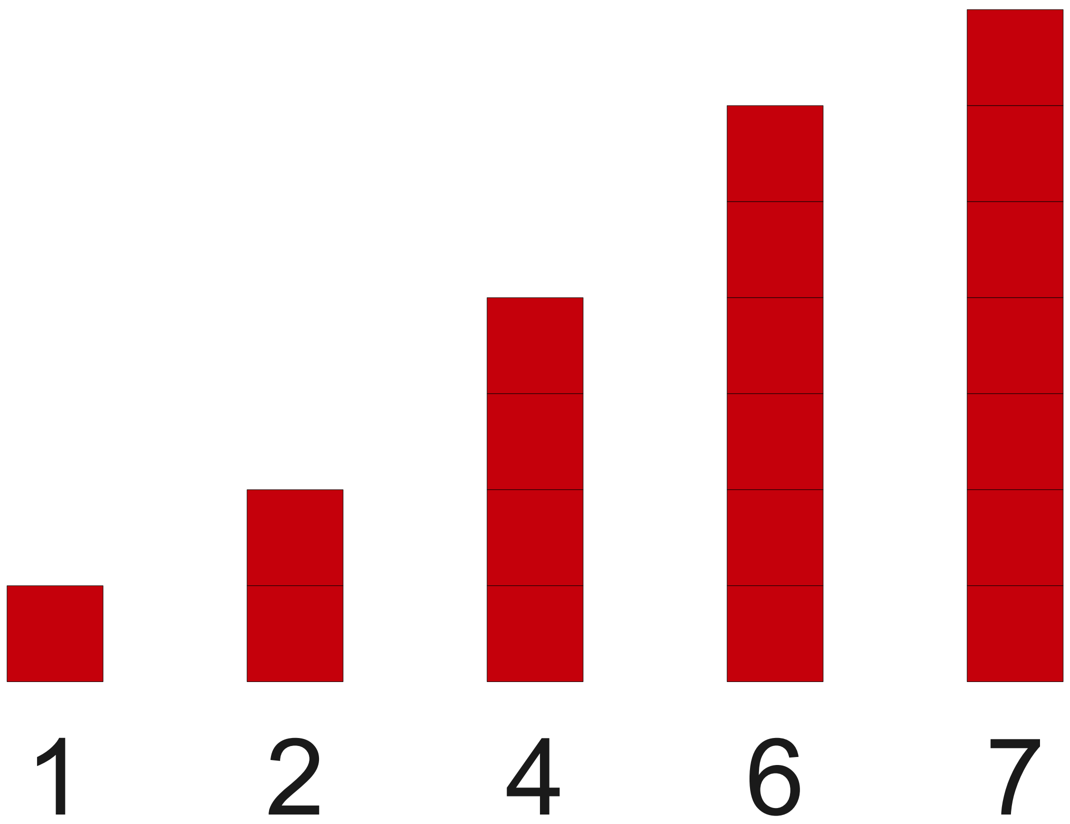
An Initial Idea: **Selection Sort**



An Initial Idea: **Selection Sort**



An Initial Idea: **Selection Sort**



Selection Sort

- Find the smallest element and move it to the first position.
- Find the second-smallest element and move it to the second position.
- (etc.)

Code for Selection Sort

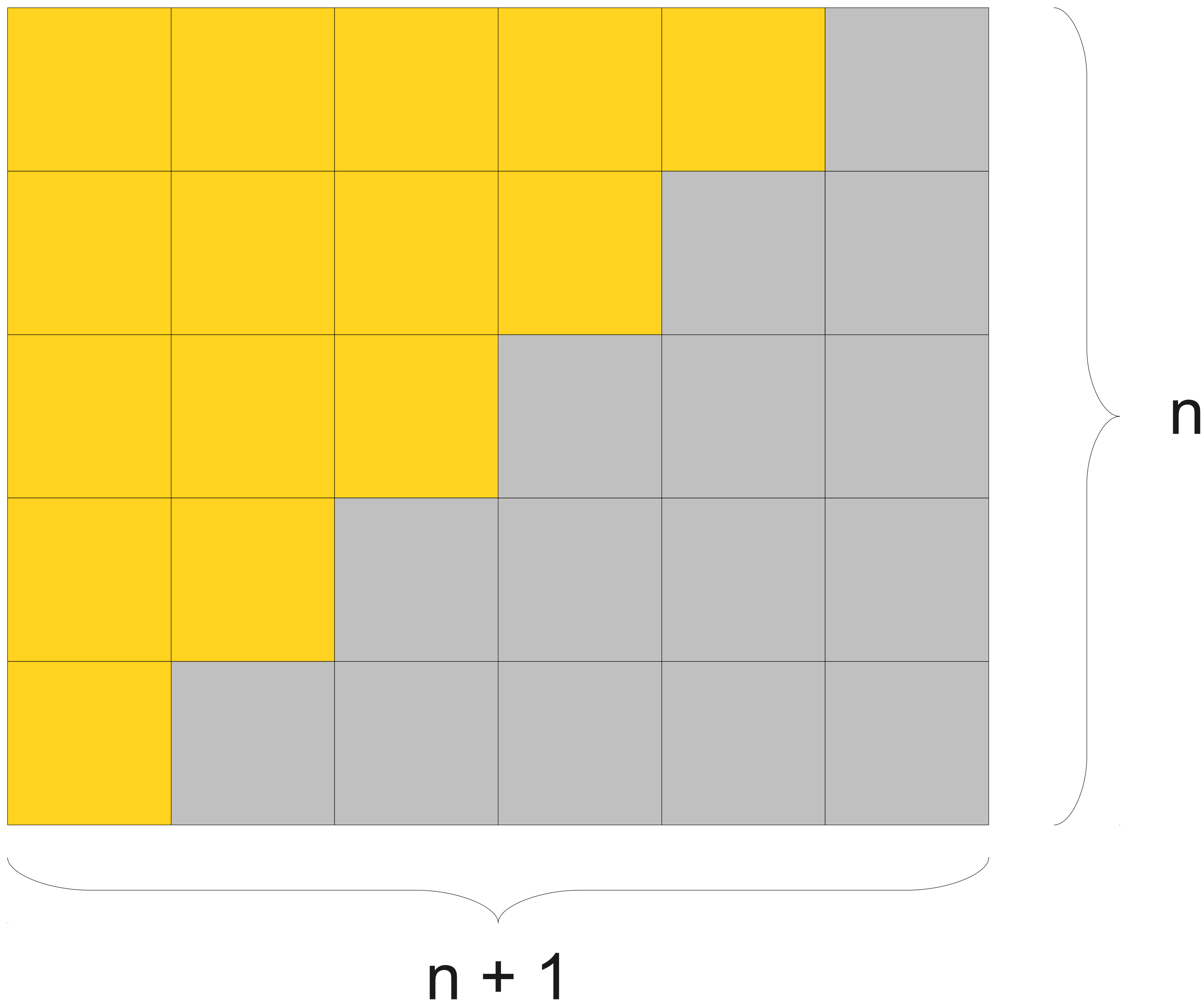
```
void selectionSort(Vector<int>& elems) {
    for (int index = 0; index < elems.size(); index++) {
        int smallestIndex = findSmallest(elems, index);
        swap(elems[index], elems[smallestIndex]);
    }
}

int findSmallest(Vector<int>& elems, int startPoint) {
    int smallestIndex = startPoint;
    for (int i = startPoint + 1; i < elems.size(); i++) {
        if (elems[i] < elems[smallestIndex])
            smallestIndex = i;
    }
    return smallestIndex;
}
```

The Complexity of Selection Sort

- Finding minimum element takes n steps.
- Finding minimum of what remains takes n – 1 steps.
- (etc.)
- Total runtime is $n + (n - 1) + \ldots + 2 + 1$.
- What is this?

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



The Complexity of Selection Sort

$O(n (n + 1) / 2)$

$= O(n (n + 1))$

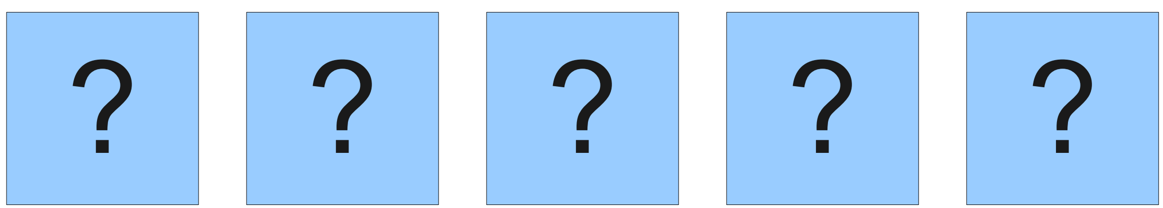
$= O(n^2 + n)$

$= O(n^2)$

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

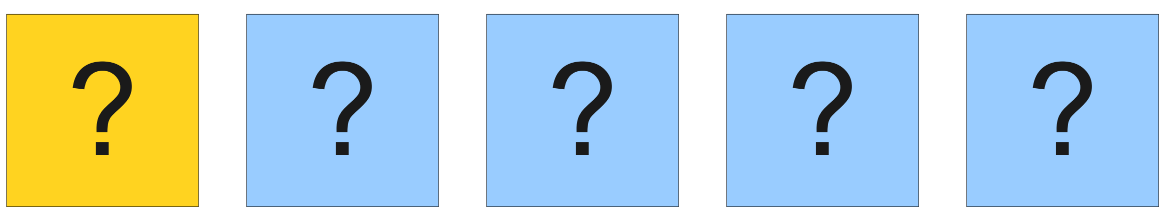
Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?



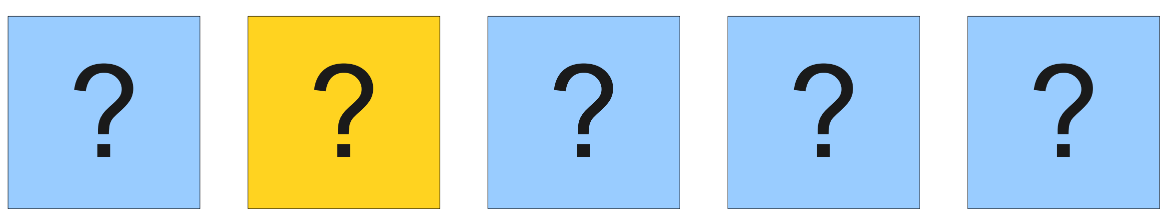
Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?



Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?

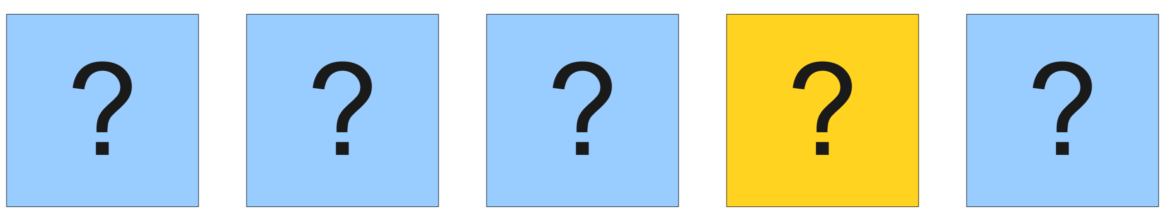


Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?

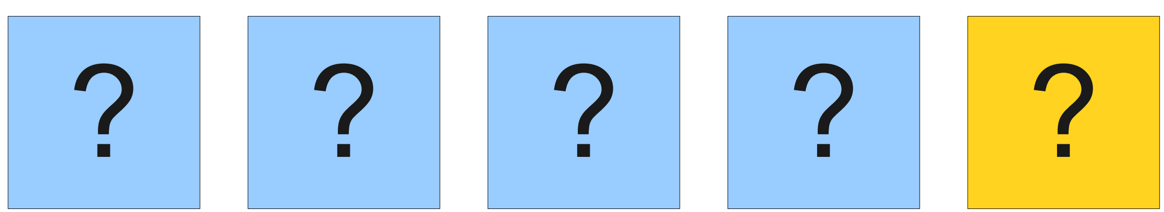
Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?



Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?

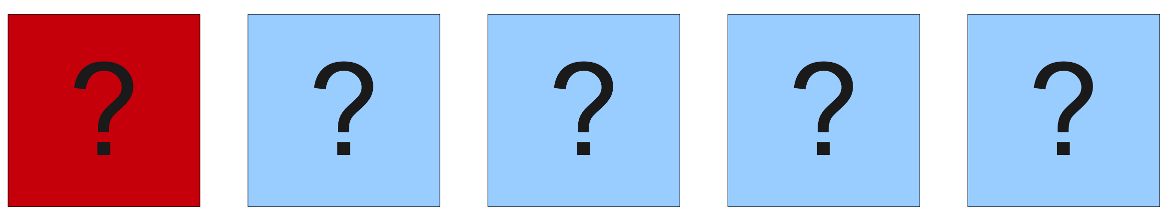


Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?

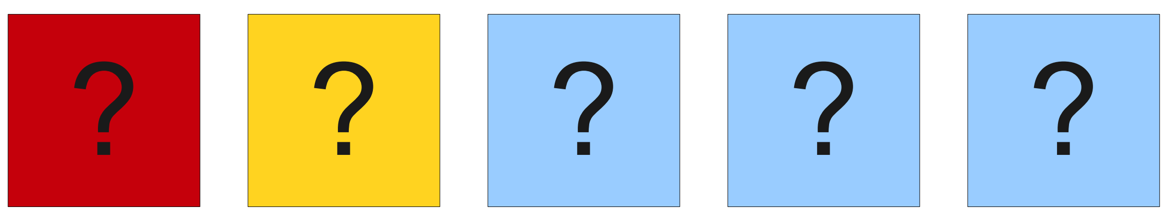
Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?



Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?

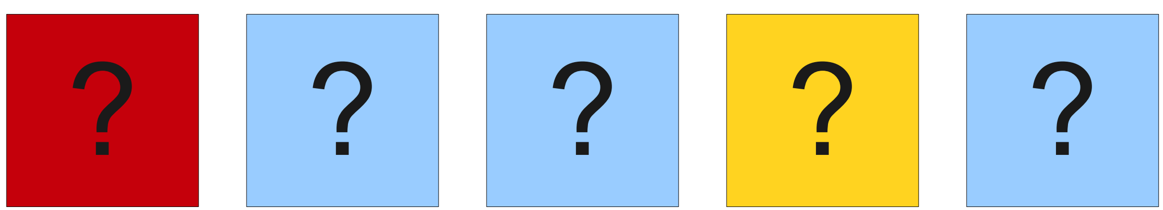


Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?

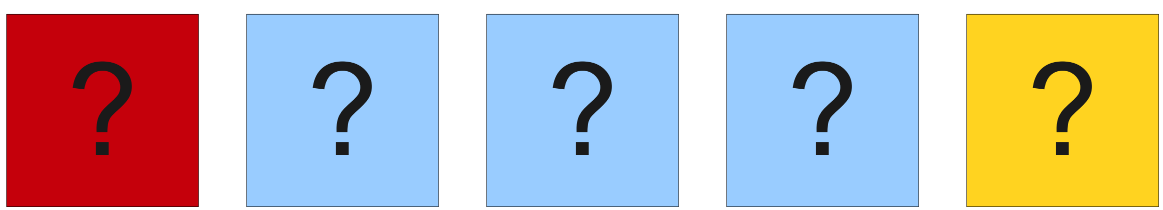
Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?



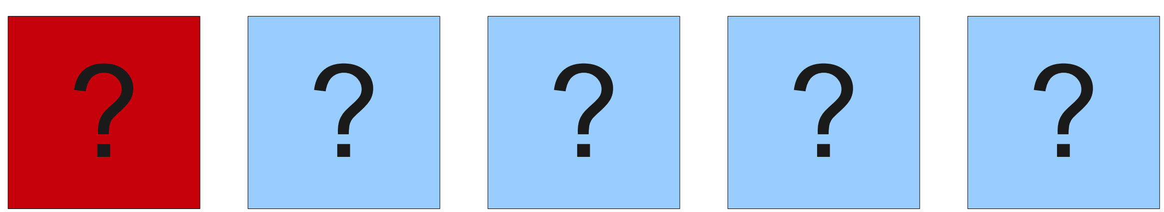
Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?



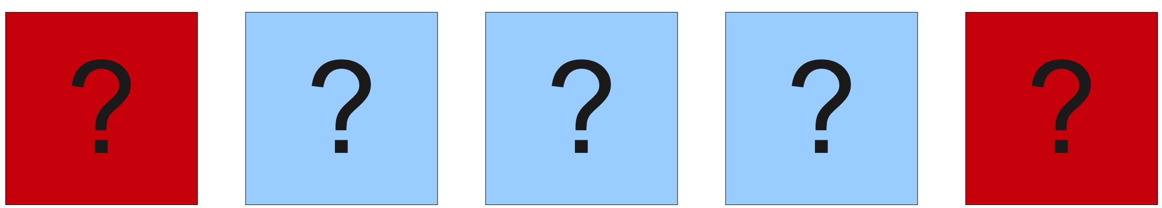
Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?



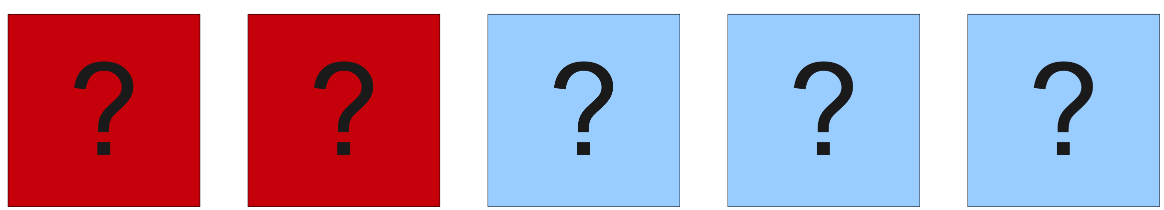
Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?



Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?



Notes on Selection Sort

- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?
- Also $O(n^2)$
- Selection sort *always* takes $O(n^2)$ time.
- Notation: Selection sort is $\Theta(n^2)$.

Thinking About $O(n^2)$

Thinking About O(n^2)

14	6	3	9	7	16	2	15

Thinking About $O(n^2)$

$$\underbrace{\begin{array}{|c|c|c|c|c|c|c|c|}\hline 14 & 6 & 3 & 9 & 7 & 16 & 2 & 15 \\ \hline\end{array}}_{T(n)}$$

Thinking About $O(n^2)$

14	6	3	9	7	16	2	15

$T(n)$

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

Thinking About $O(n^2)$

14	6	3	9	7	16	2	15

$$\underbrace{\qquad\qquad\qquad}_{T(n)}$$

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

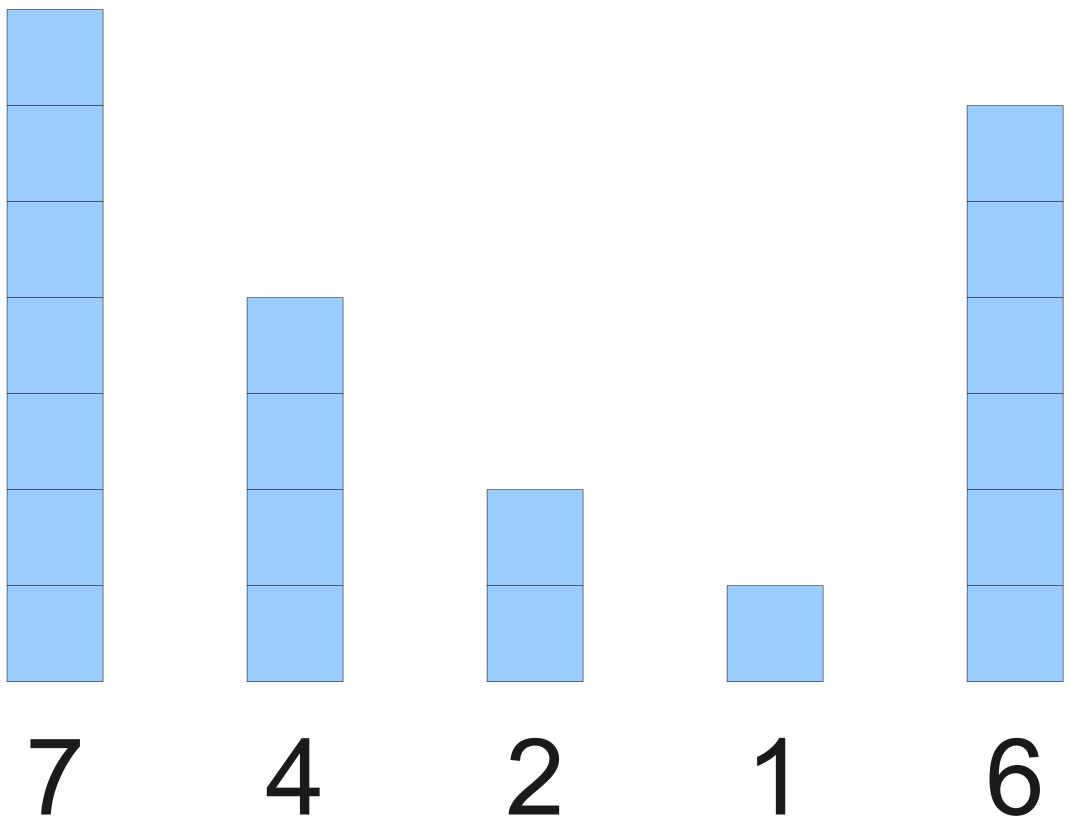
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{T(2n) \approx 4T(n)}$$

Selection Sort Times

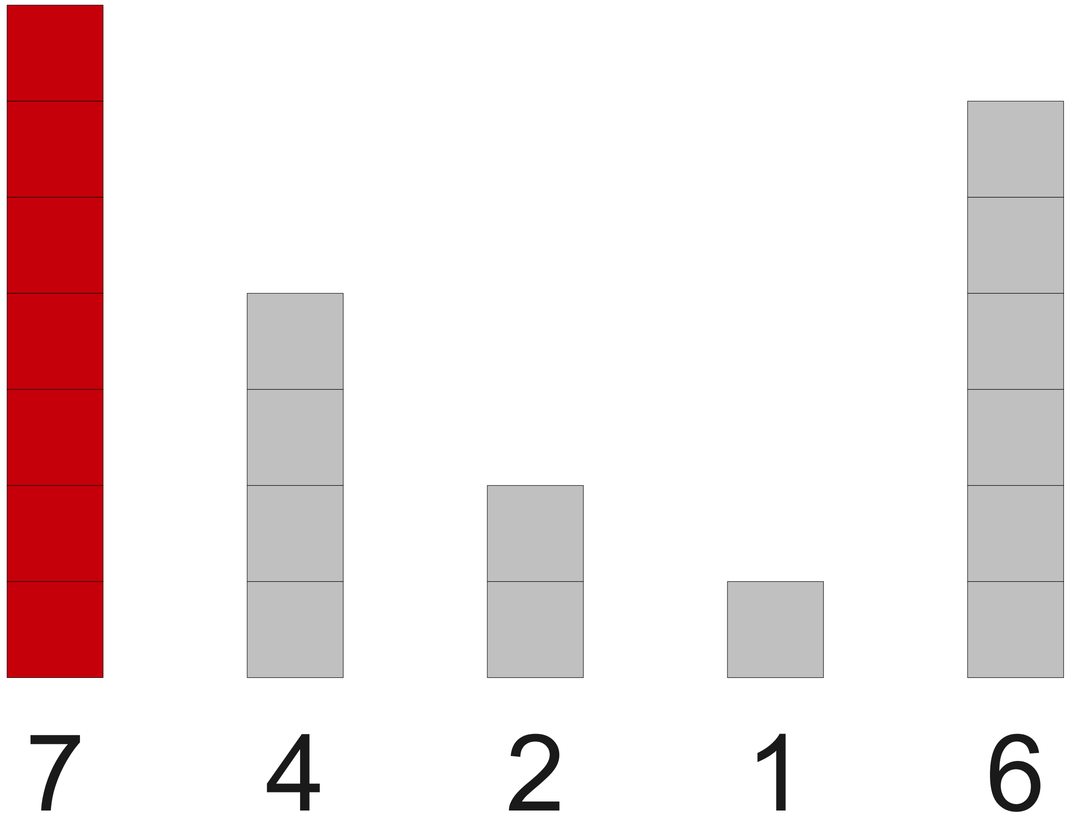
Size	Selection Sort
10000	0.304
20000	1.218
30000	2.790
40000	4.646
50000	7.395
60000	10.584
70000	14.149
80000	18.674
90000	23.165

Another Idea: **Insertion Sort**

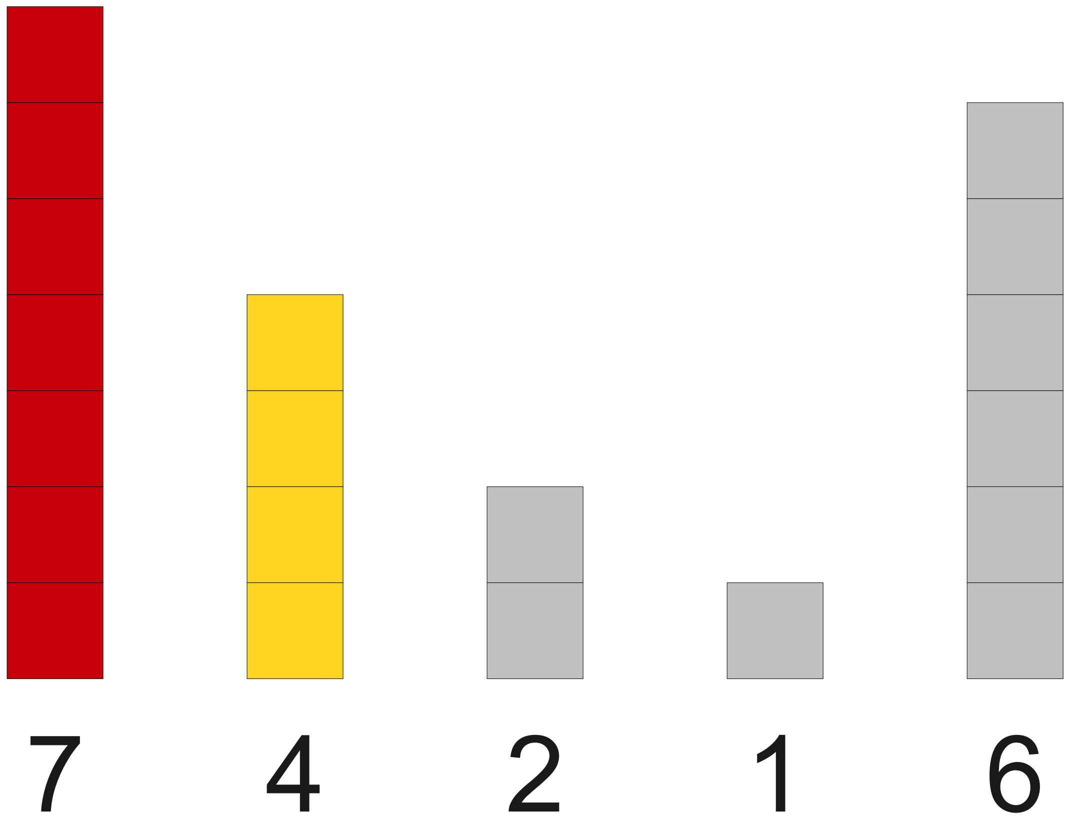
Another Idea: **Insertion Sort**



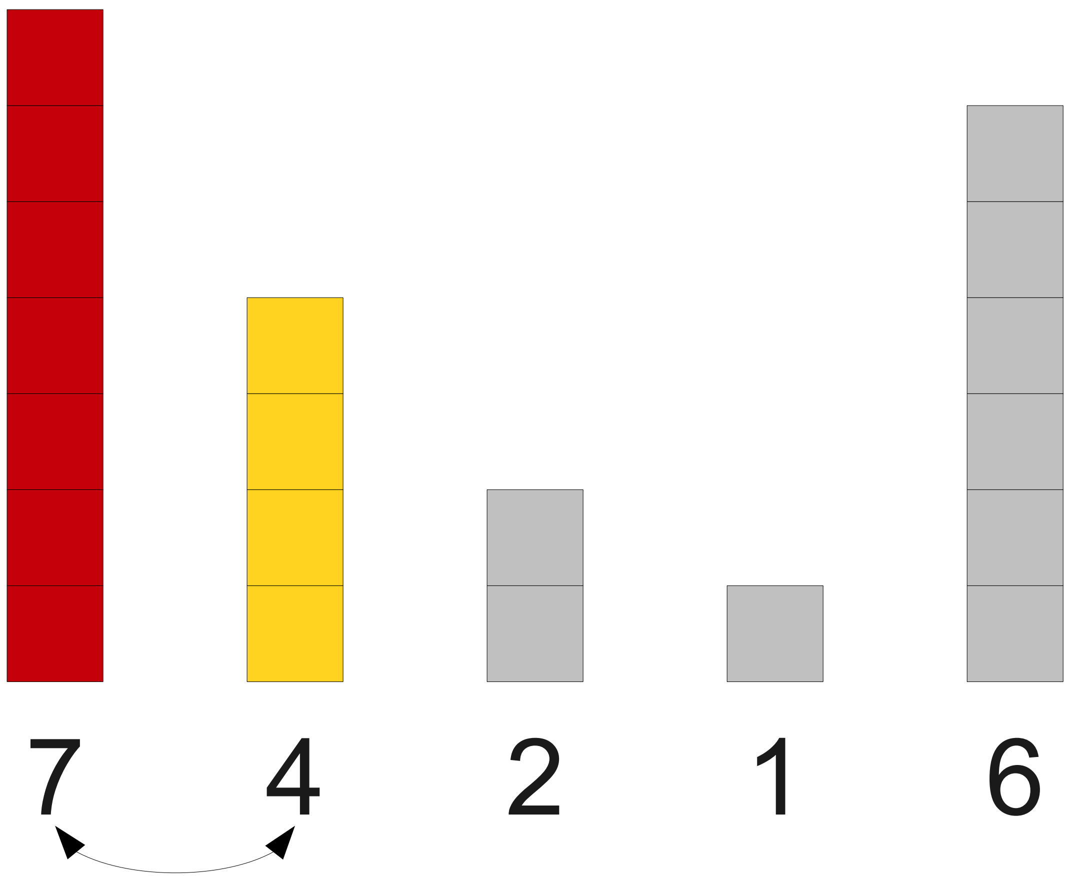
Another Idea: **Insertion Sort**



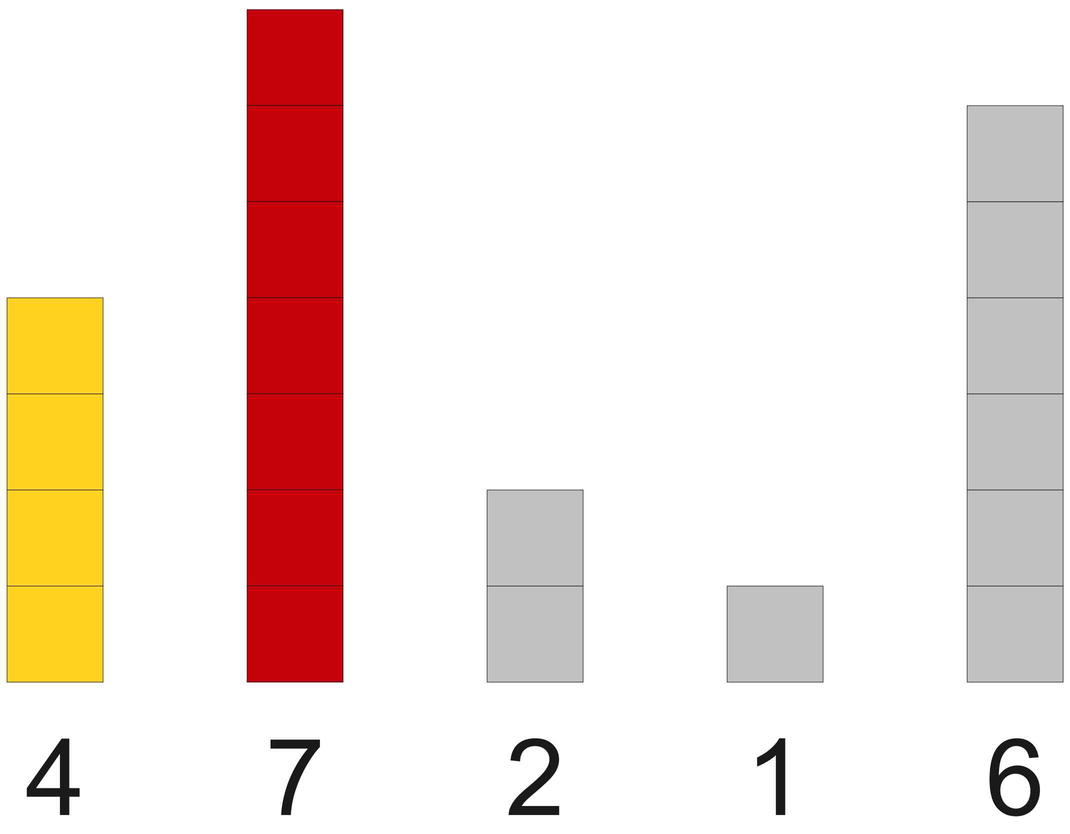
Another Idea: **Insertion Sort**



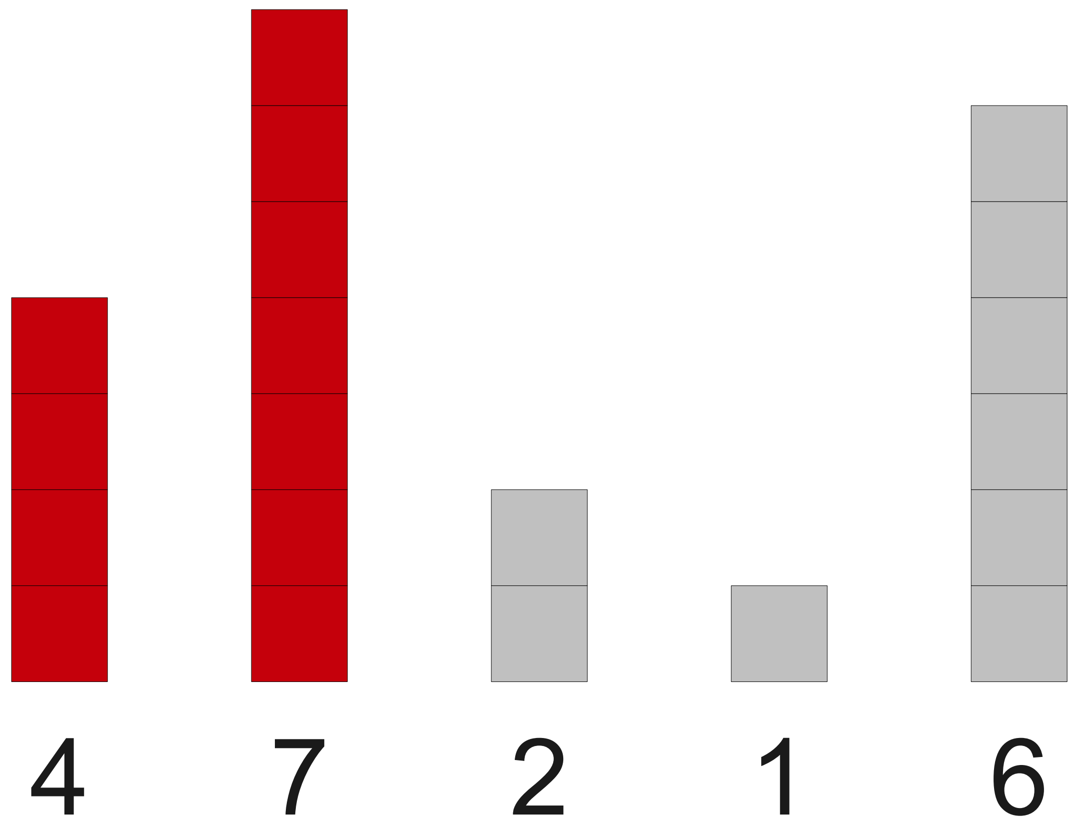
Another Idea: **Insertion Sort**



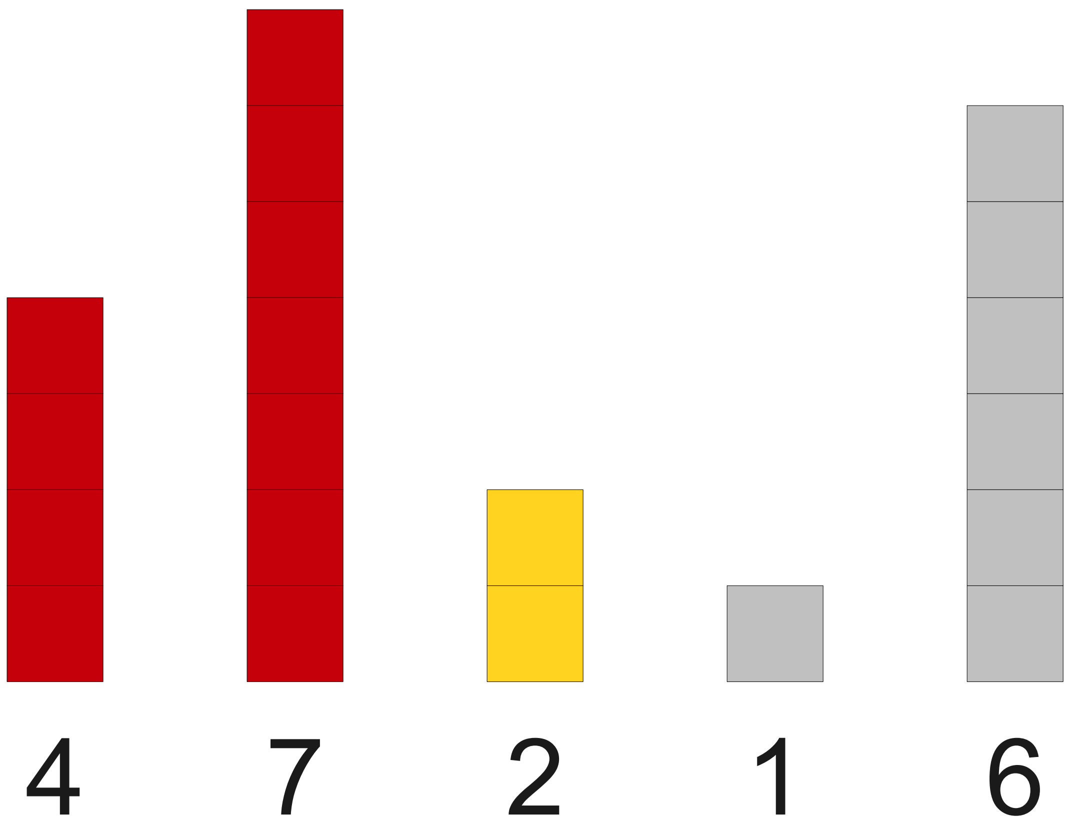
Another Idea: **Insertion Sort**



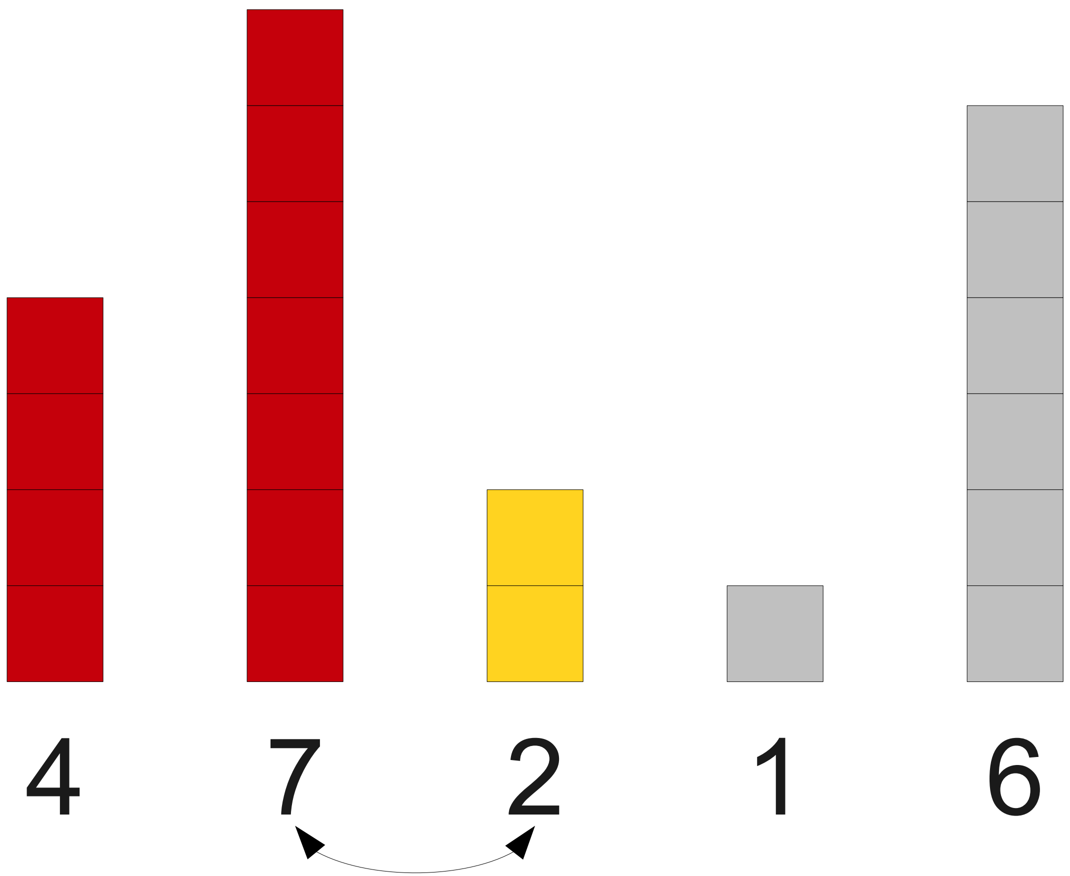
Another Idea: **Insertion Sort**



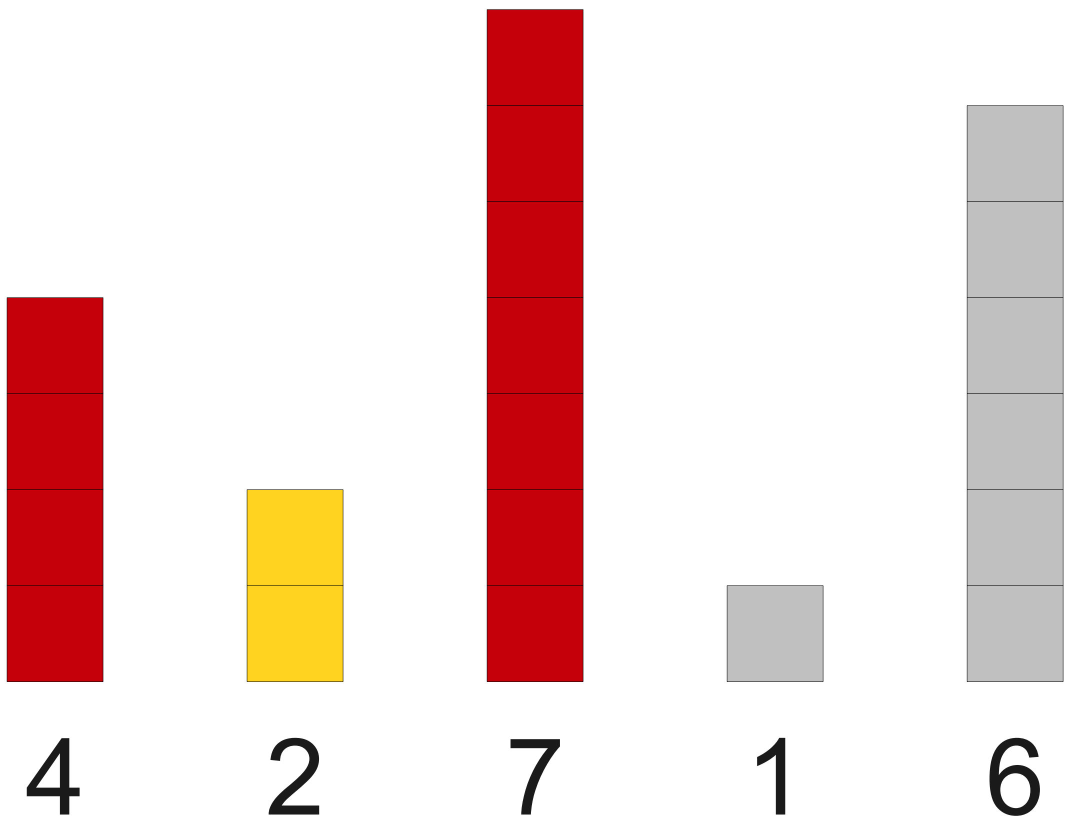
Another Idea: **Insertion Sort**



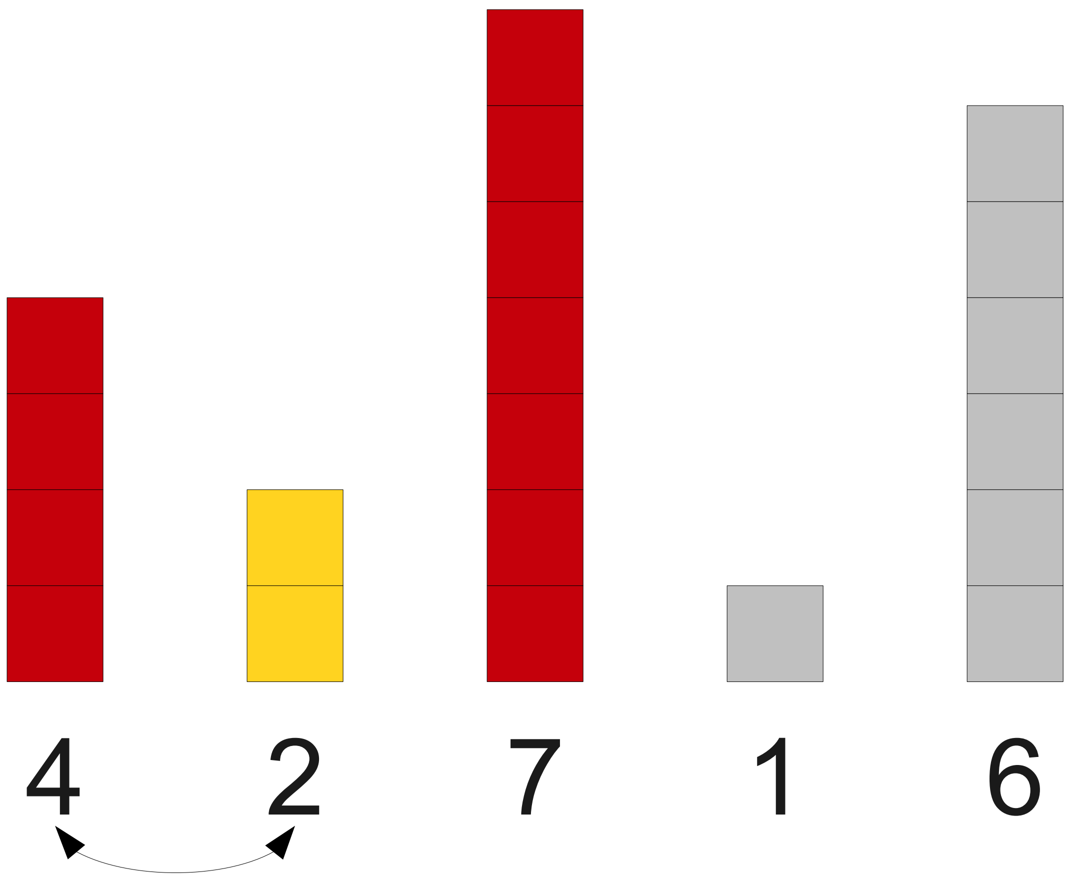
Another Idea: **Insertion Sort**



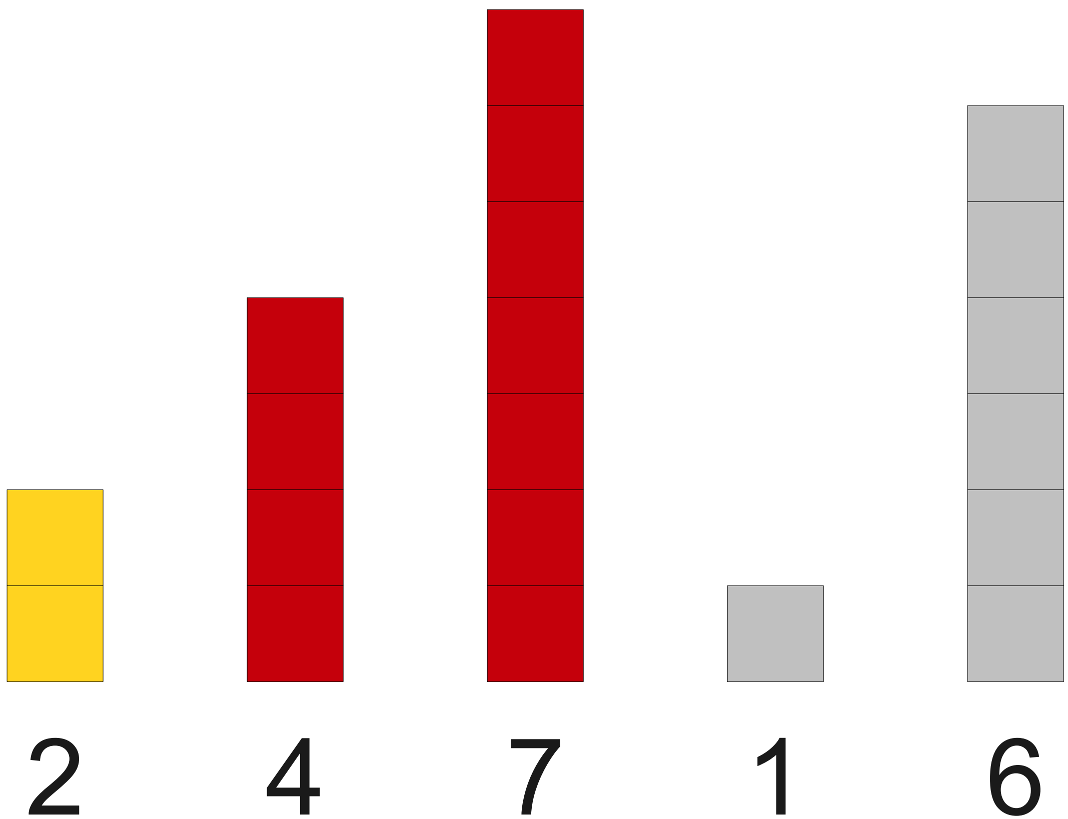
Another Idea: **Insertion Sort**



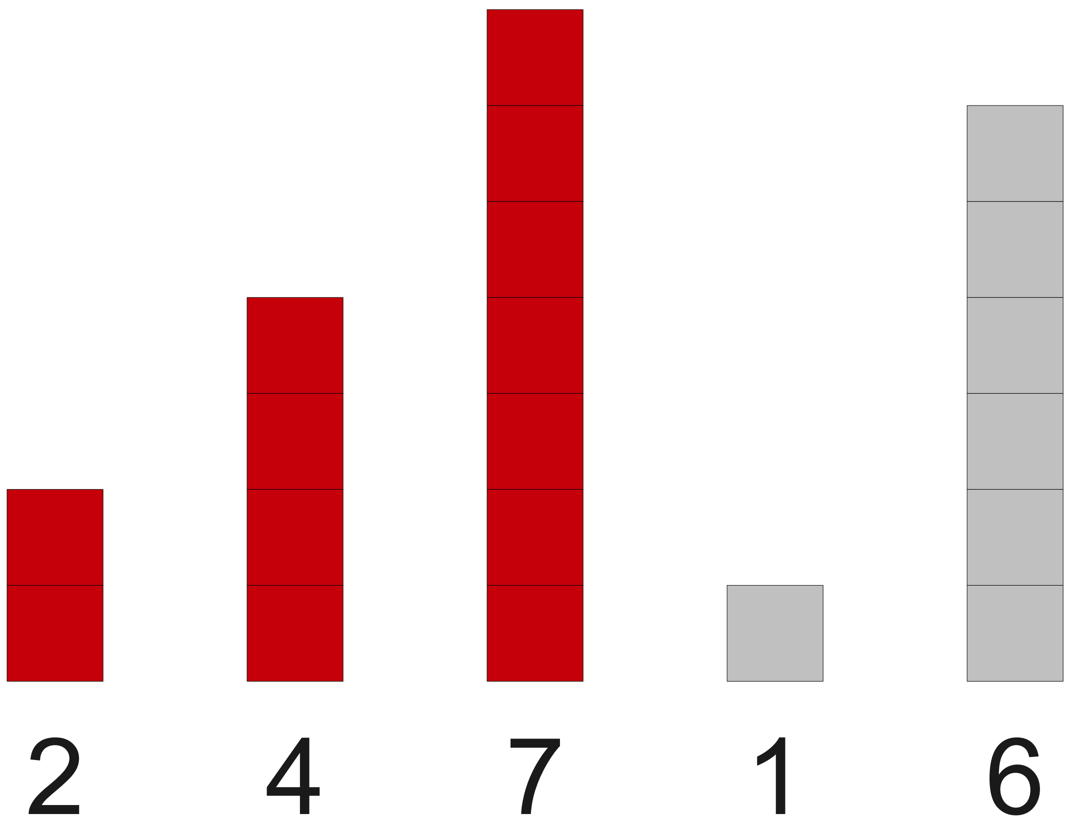
Another Idea: **Insertion Sort**



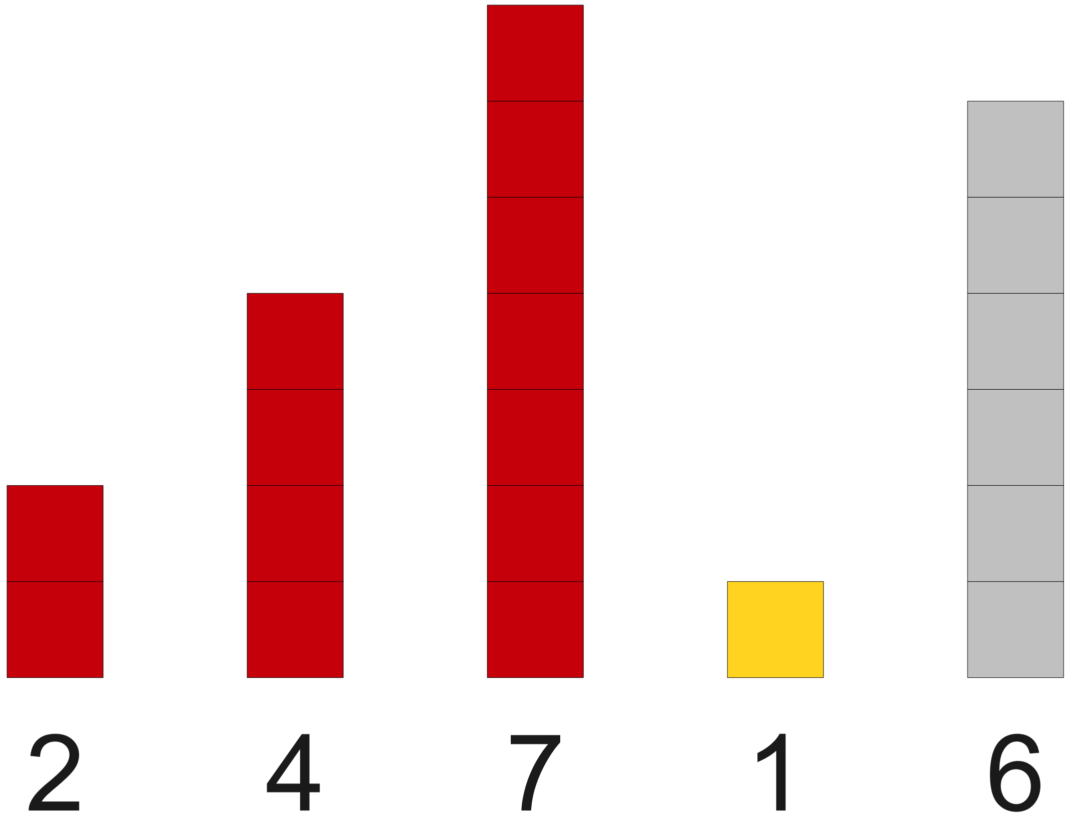
Another Idea: **Insertion Sort**



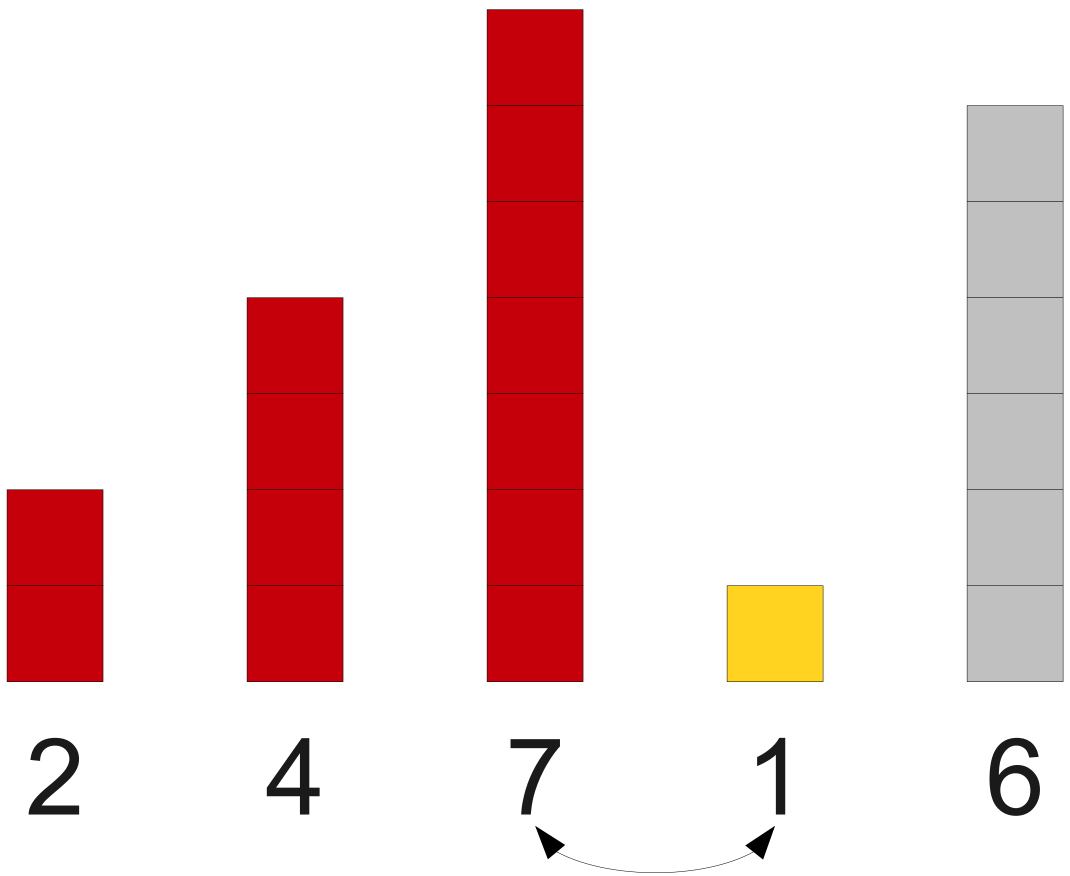
Another Idea: **Insertion Sort**



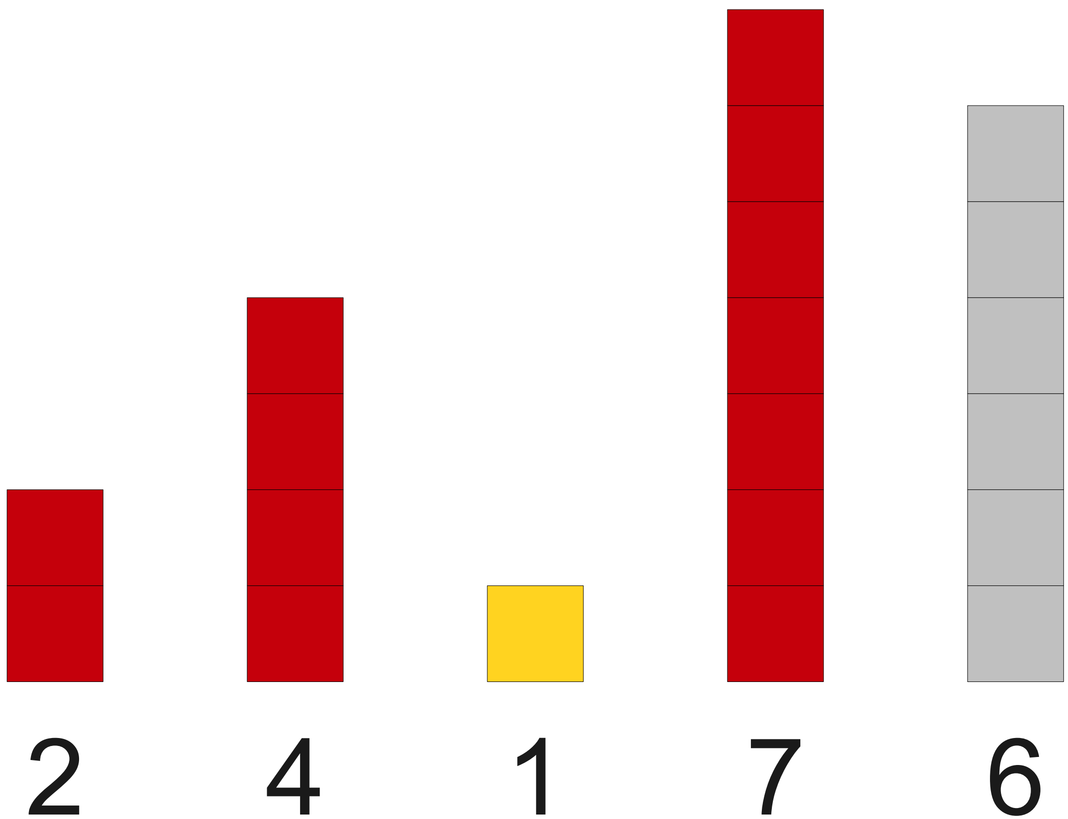
Another Idea: Insertion Sort



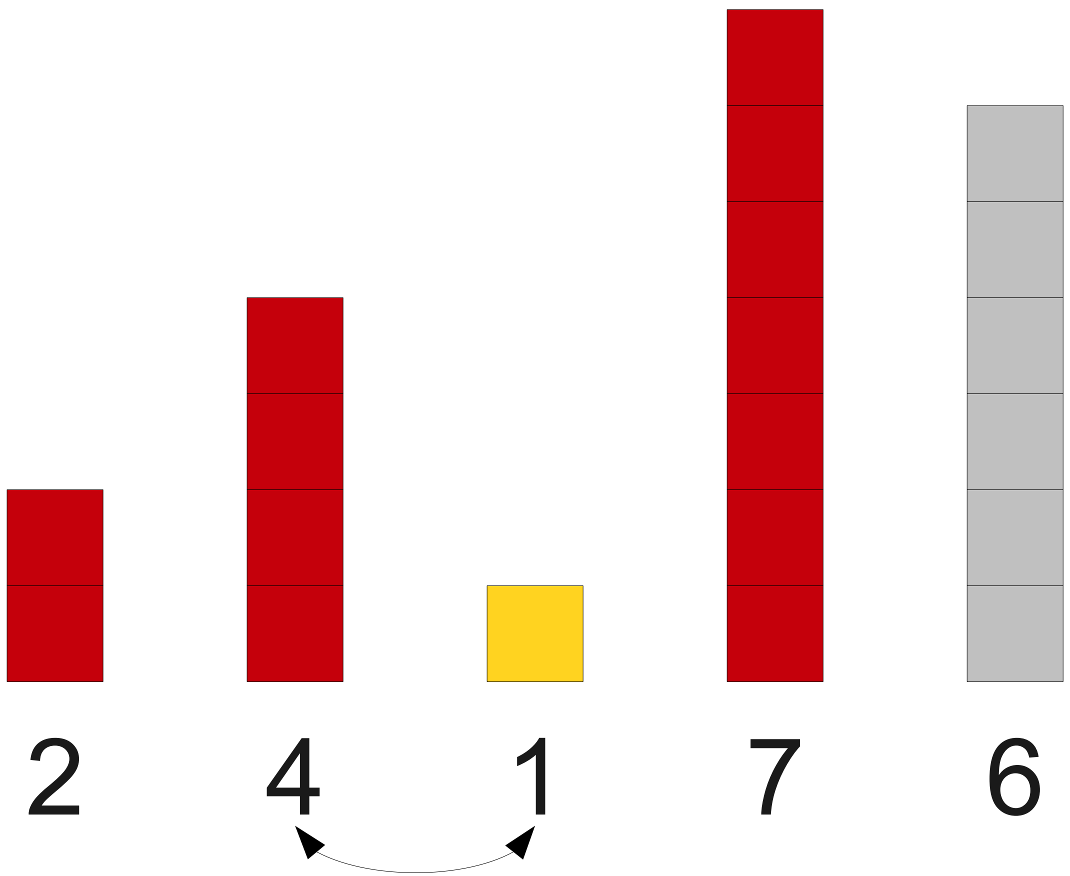
Another Idea: **Insertion Sort**



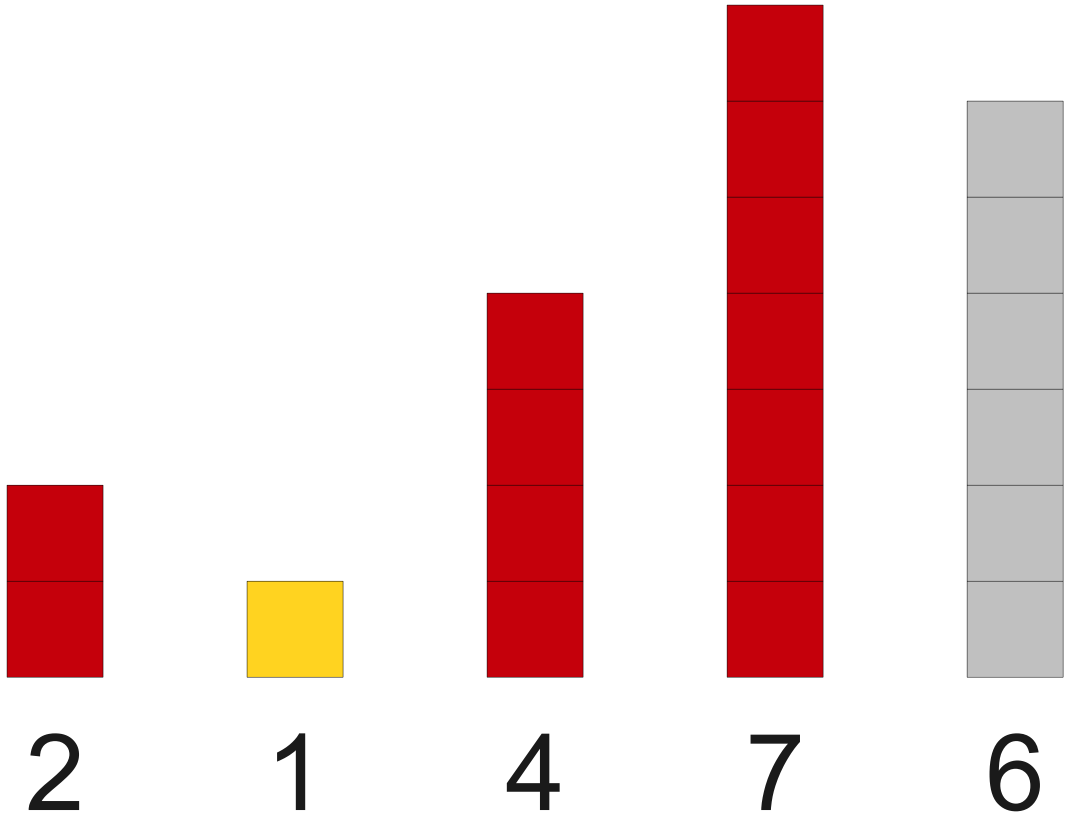
Another Idea: **Insertion Sort**



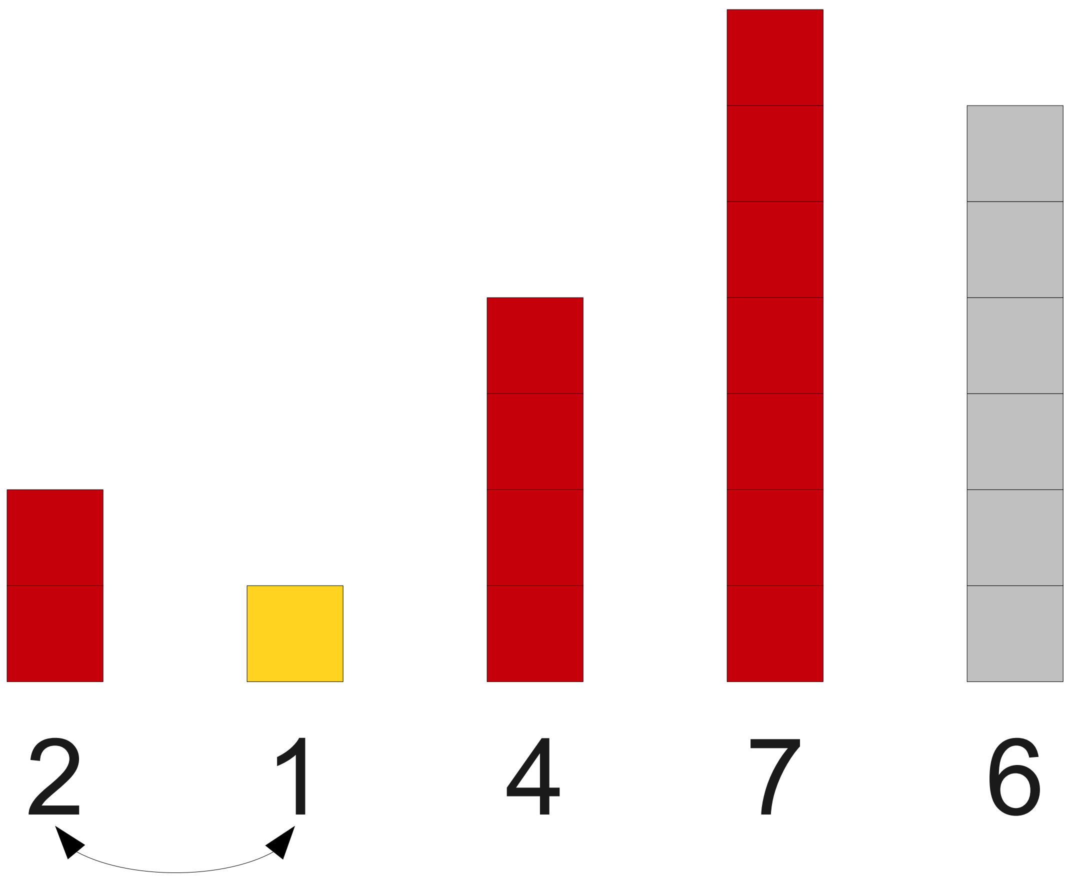
Another Idea: **Insertion Sort**



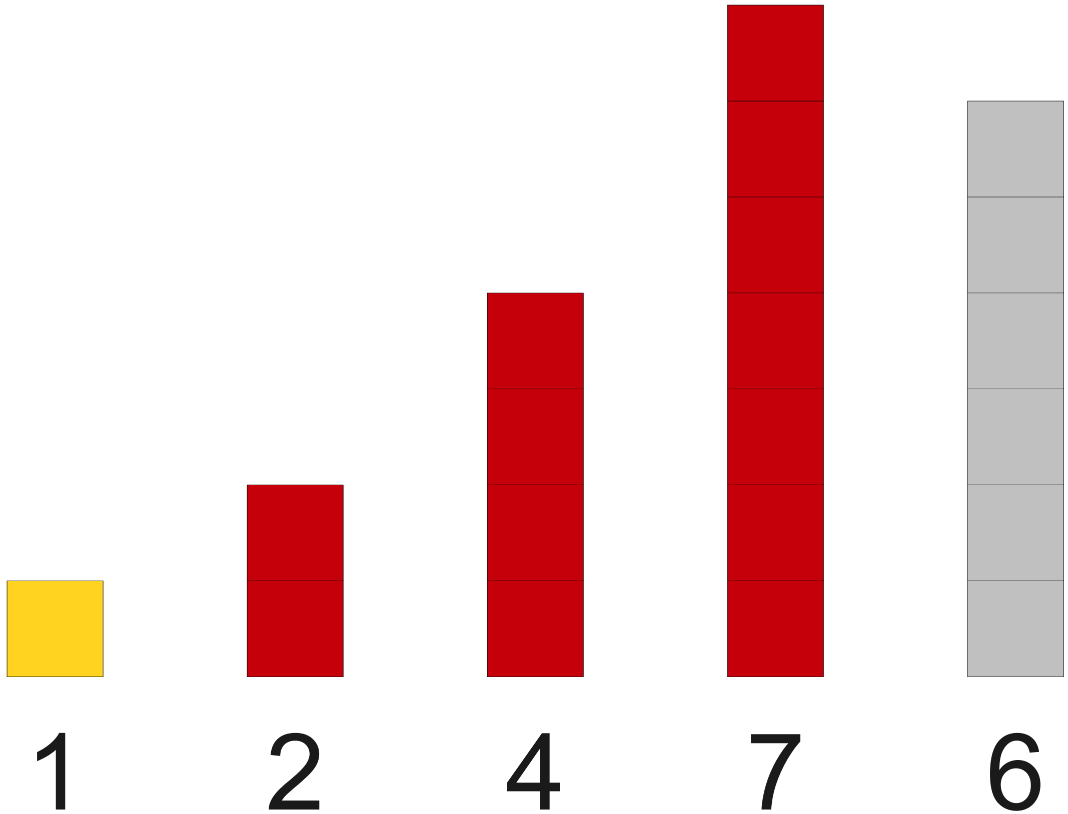
Another Idea: **Insertion Sort**



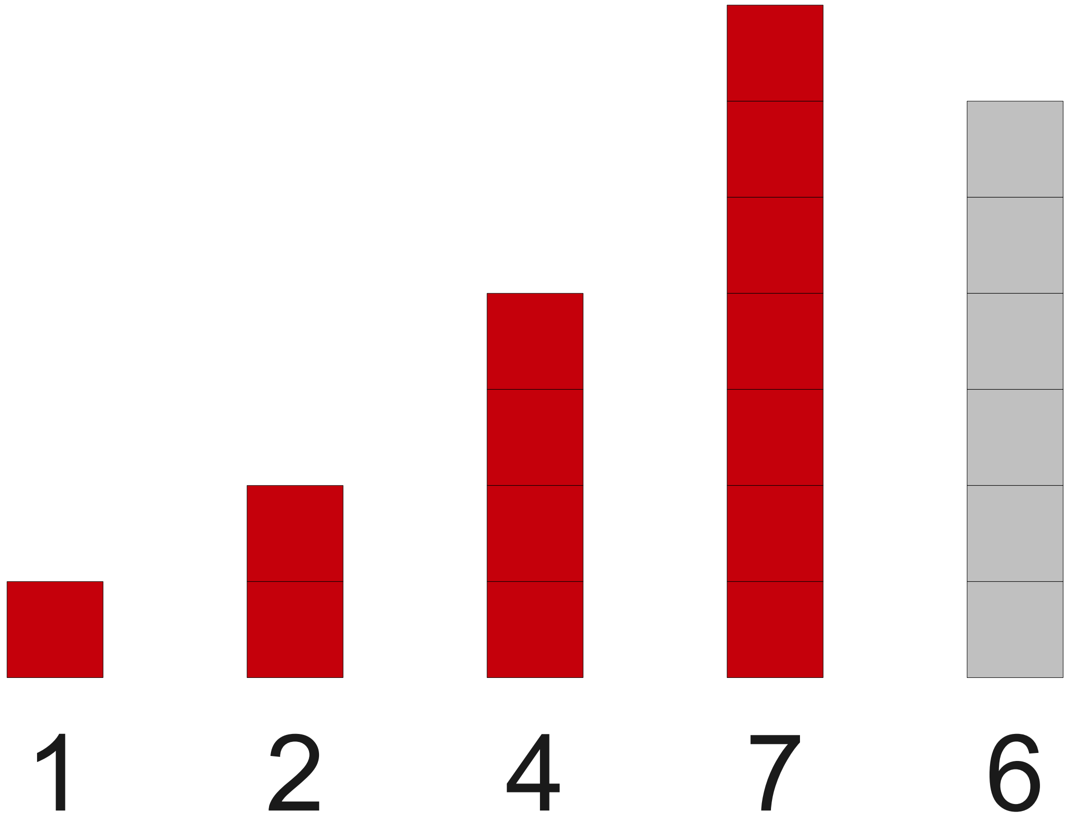
Another Idea: **Insertion Sort**



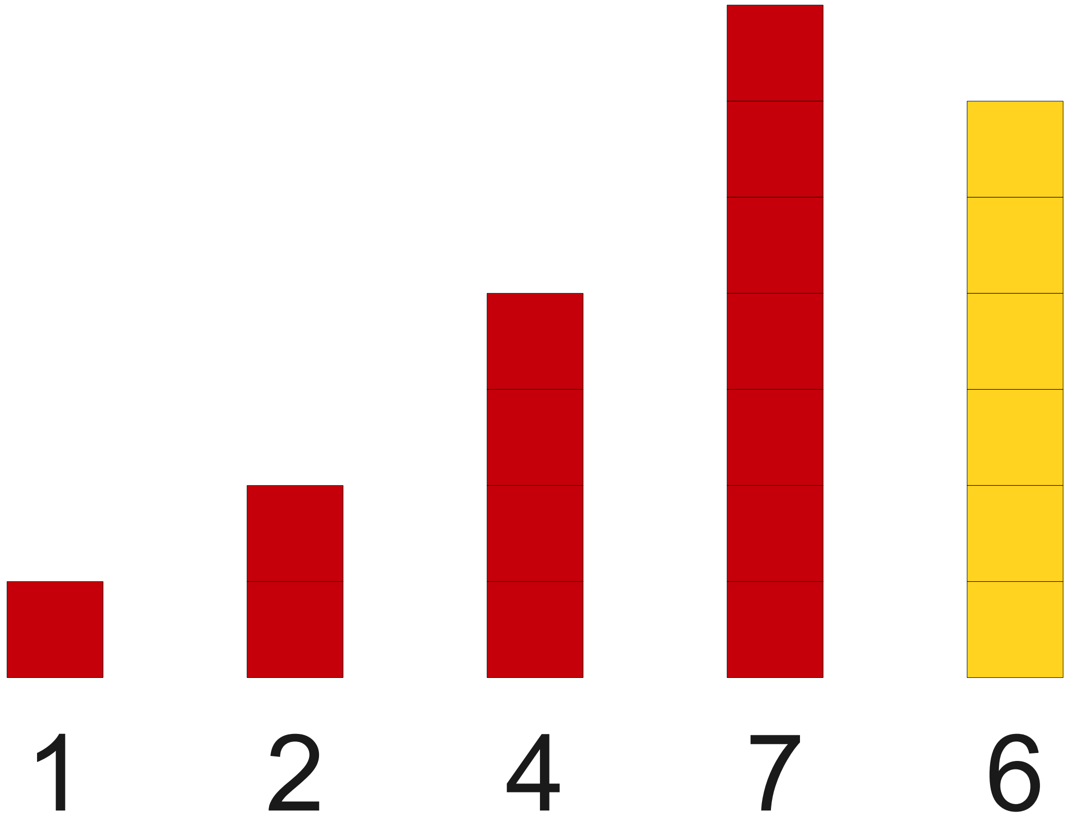
Another Idea: **Insertion Sort**



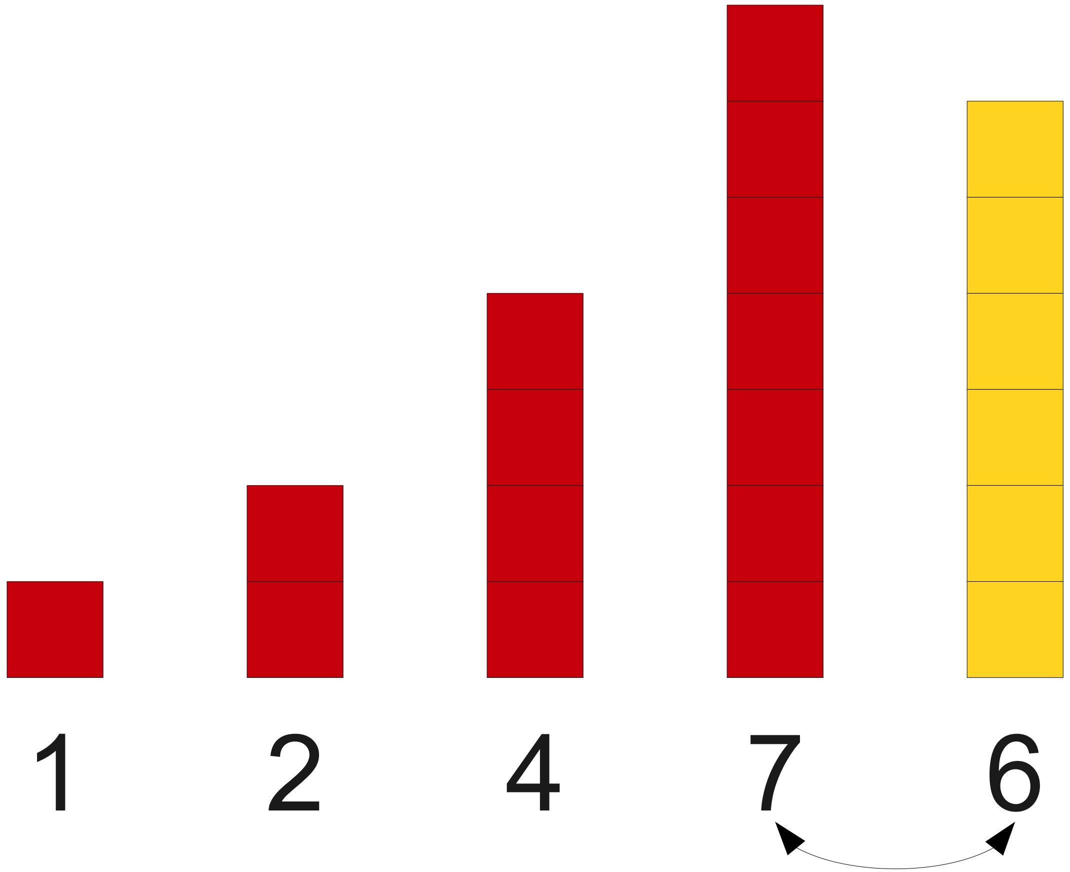
Another Idea: **Insertion Sort**



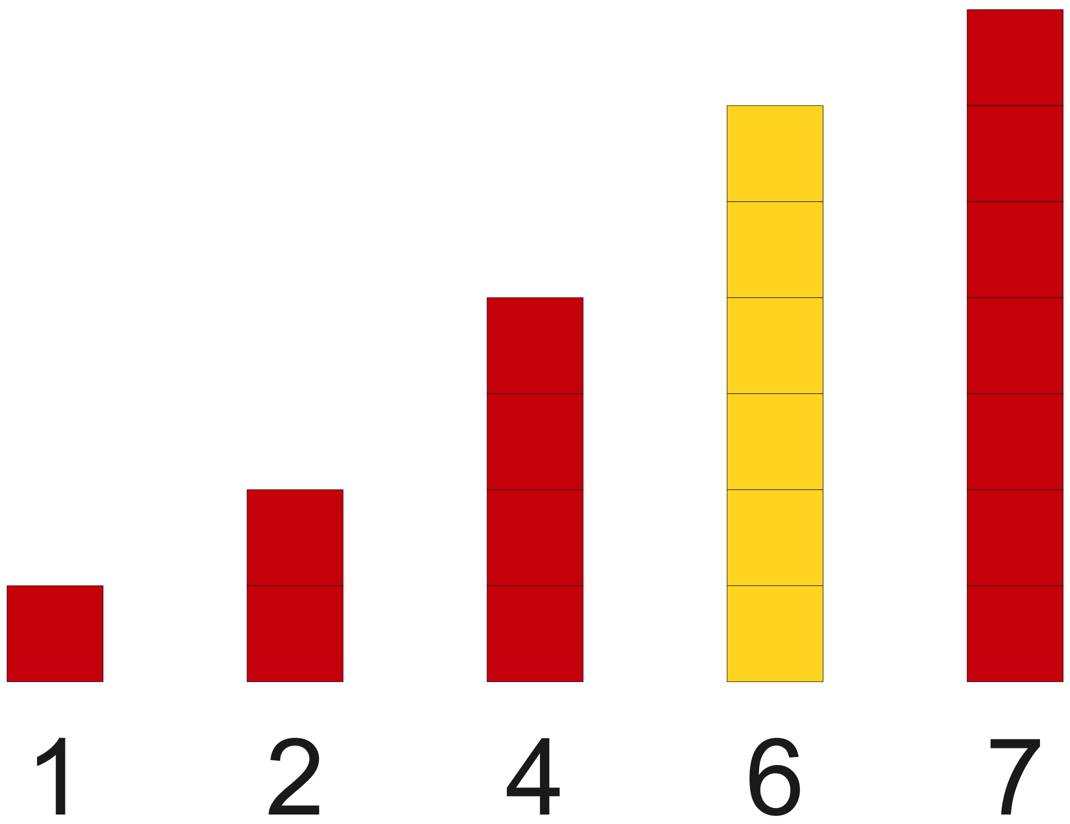
Another Idea: **Insertion Sort**



Another Idea: **Insertion Sort**



Another Idea: **Insertion Sort**



Another Idea: **Insertion Sort**