

Algorithmic Analysis and Sorting, Part One

CS106B
Winter 2009-2010

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.

Another Idea: **Instruction Count**

```
bool LinearSearch(string& str, char ch)
{
    bool found = false;

    for (int i = 0; i < str.length(); i++)
        if (str[i] == ch)
            found = true;

    return found;
}
```

$$\text{Instruction Count} = k_0 n + k_1$$

What can we learn from $k_0n + k_1$?

- Runtime increases as function of input.
 - Longer inputs take more time.
- Runtime increases **linearly** as function of input.
 - Longer inputs take **proportionally** more time.
- But we still can't predict the runtime!
 - Same reasoning as before.

Big Observations

- Constants don't matter
 - Whether runtime is $4n + 10$ or $100n + 137$, runtime is still proportional to input size.
 - Difficult or impossible to accurately compute 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.
- Examples:
 - $4n + 4 = O(n)$
 - $137n + 271 = O(n)$
 - $n^2 + 3n + 4 = O(n^2)$
 - $2^n + n^3 = O(2^n)$

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;
}
```

$O(n)$

Counting Duplicates

```
int CountDuplicates(Vector<int>& v)
{
    int result = 0;

    for (int i = 0; i < v.size(); i++)
        for (int j = i + 1; j < v.size(); j++)
            if (v[i] == v[j])
                result ++;

    return result;
}
```

Counting Duplicates

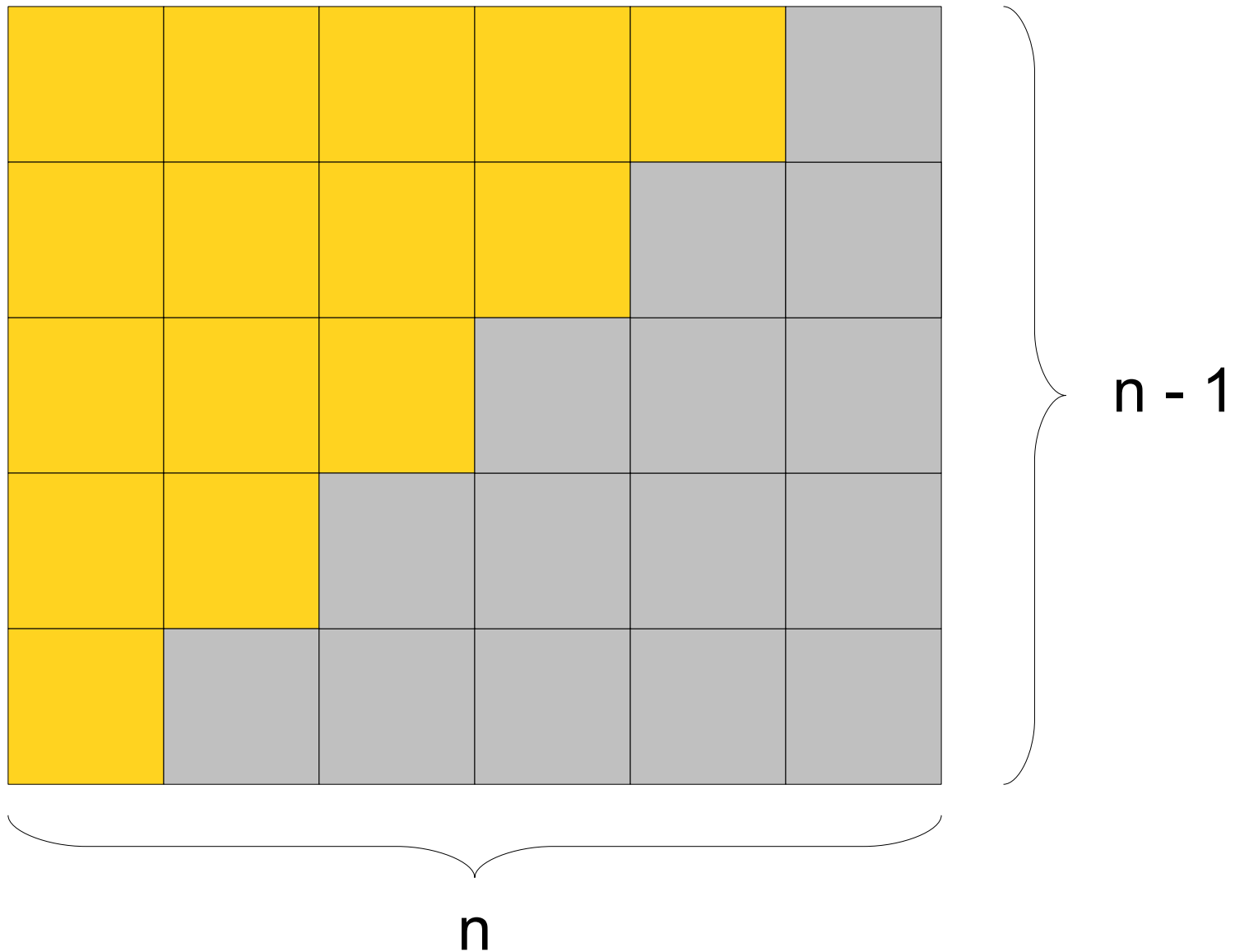
```
int CountDuplicates(Vector<int>& v)
{
    int result = 0;

    for (int i = 0; i < v.size(); i++)
        for (int j = i + 1; j < v.size(); j++)
            if (v[i] == v[j])
                result ++;

    return result;
}
```

How do we analyze this algorithm?

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



Counting Duplicates

```
int CountDuplicates(Vector<int>& v)
{
    int result = 0;

    for (int i = 0; i < v.size(); i++)
        for (int j = i + 1; j < v.size(); j++)
            if (v[i] == v[j])
                result ++;

    return result;
}
```

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

Counting Duplicates

```
int CountDuplicates(Vector<int>& v)
{
    int result = 0;

    for (int i = 0; i < v.size(); i++)
        for (int j = i + 1; j < v.size(); j++)
            if (v[i] == v[j])
                result ++;

    return result;
}
```

$O(0.5 n(n-1))$

Counting Duplicates

```
int CountDuplicates(Vector<int>& v)
{
    int result = 0;

    for (int i = 0; i < v.size(); i++)
        for (int j = i + 1; j < v.size(); j++)
            if (v[i] == v[j])
                result ++;

    return result;
}
```

$O(0.5n^2 - 0.5n)$

Counting Duplicates

```
int CountDuplicates(Vector<int>& v)
{
    int result = 0;

    for (int i = 0; i < v.size(); i++)
        for (int j = i + 1; j < v.size(); j++)
            if (v[i] == v[j])
                result ++;

    return result;
}
```

$O(0.5n^2)$

Counting Duplicates

```
int CountDuplicates(Vector<int>& v)
{
    int result = 0;

    for (int i = 0; i < v.size(); i++)
        for (int j = i + 1; j < v.size(); j++)
            if (v[i] == v[j])
                result ++;

    return result;
}
```

$O(n^2)$

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)

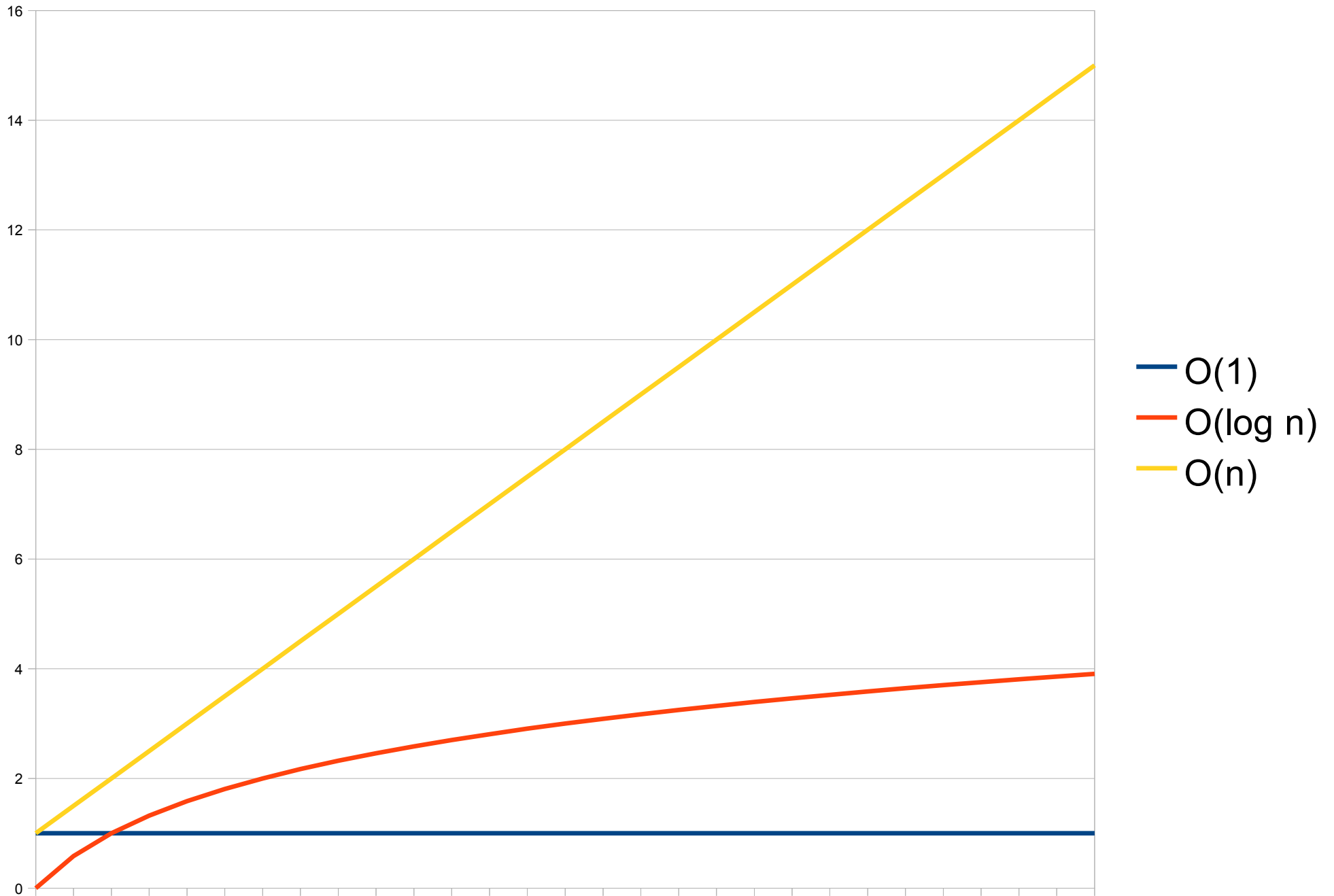
What can big-O tell us?

- 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.

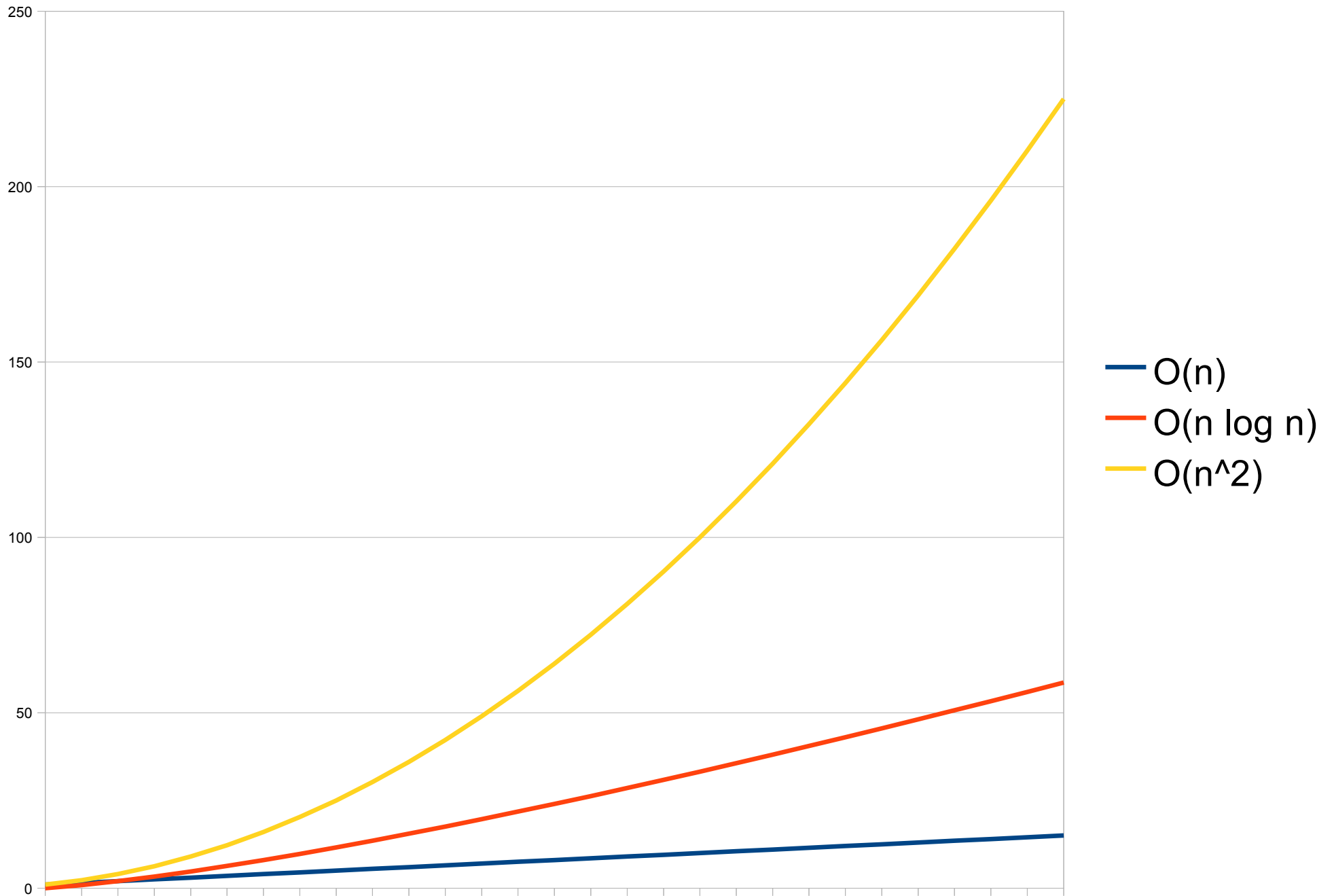
What *can't* big-O tell us?

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

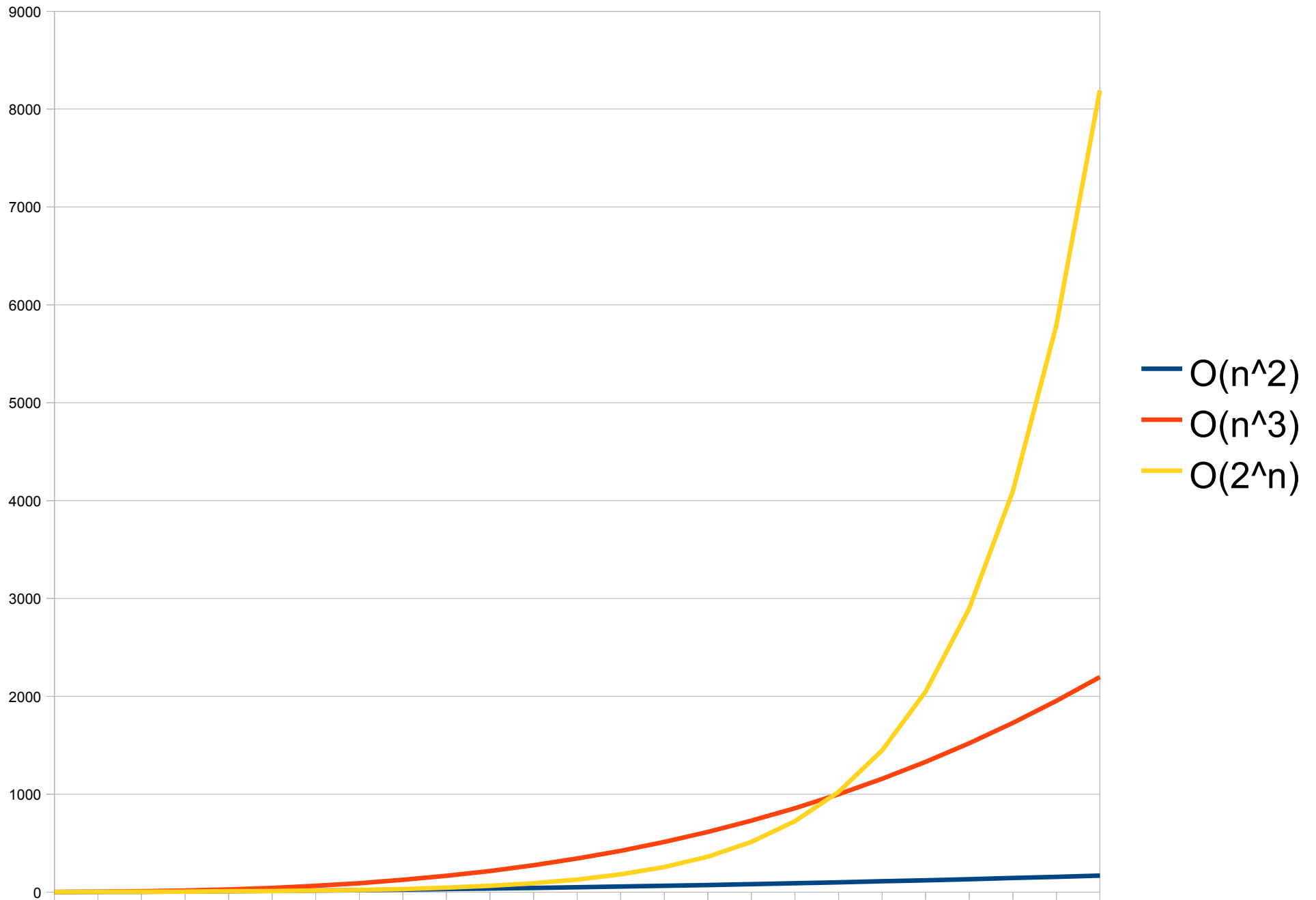
Growth Rates, Part One



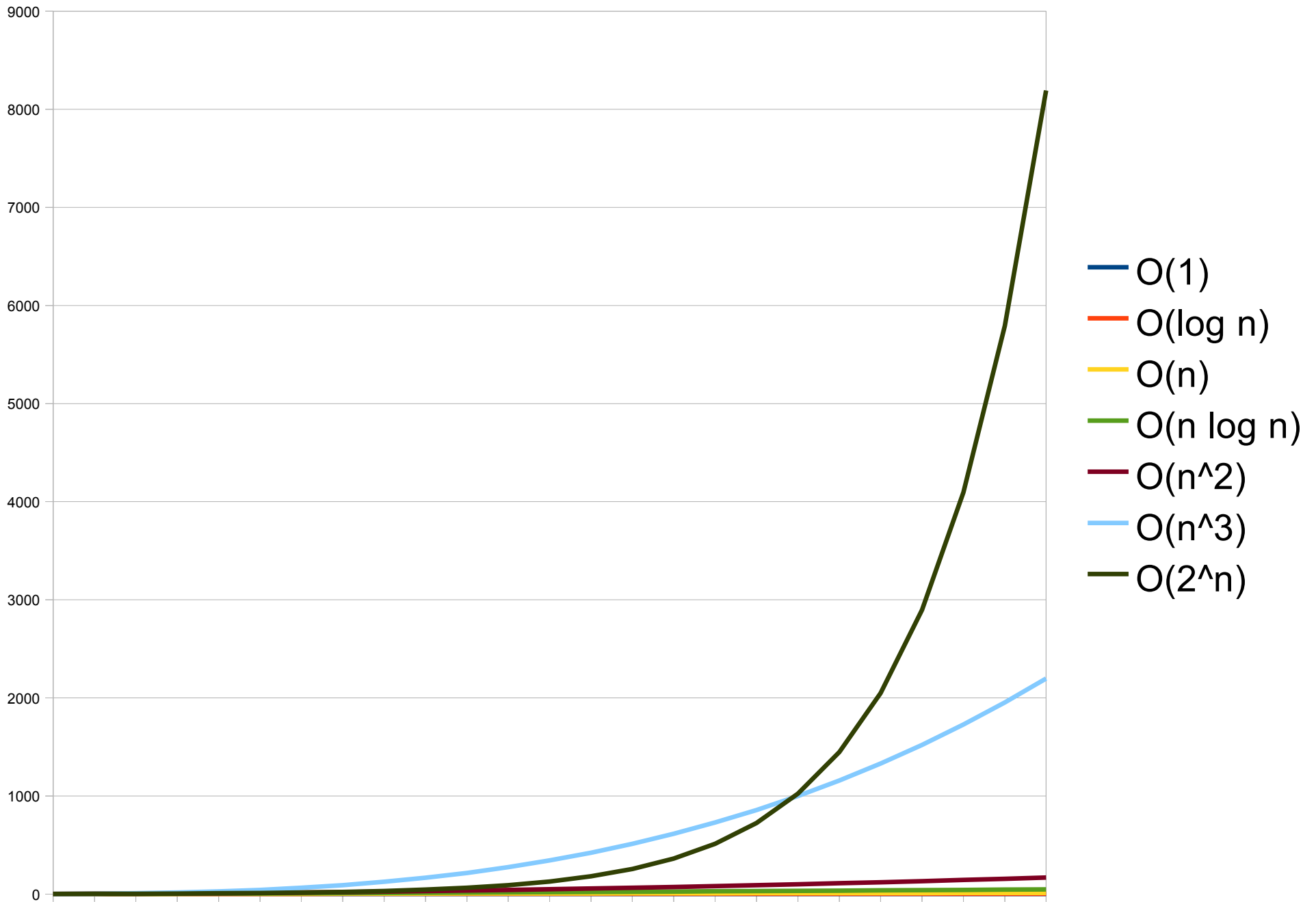
Growth Rates, Part Two



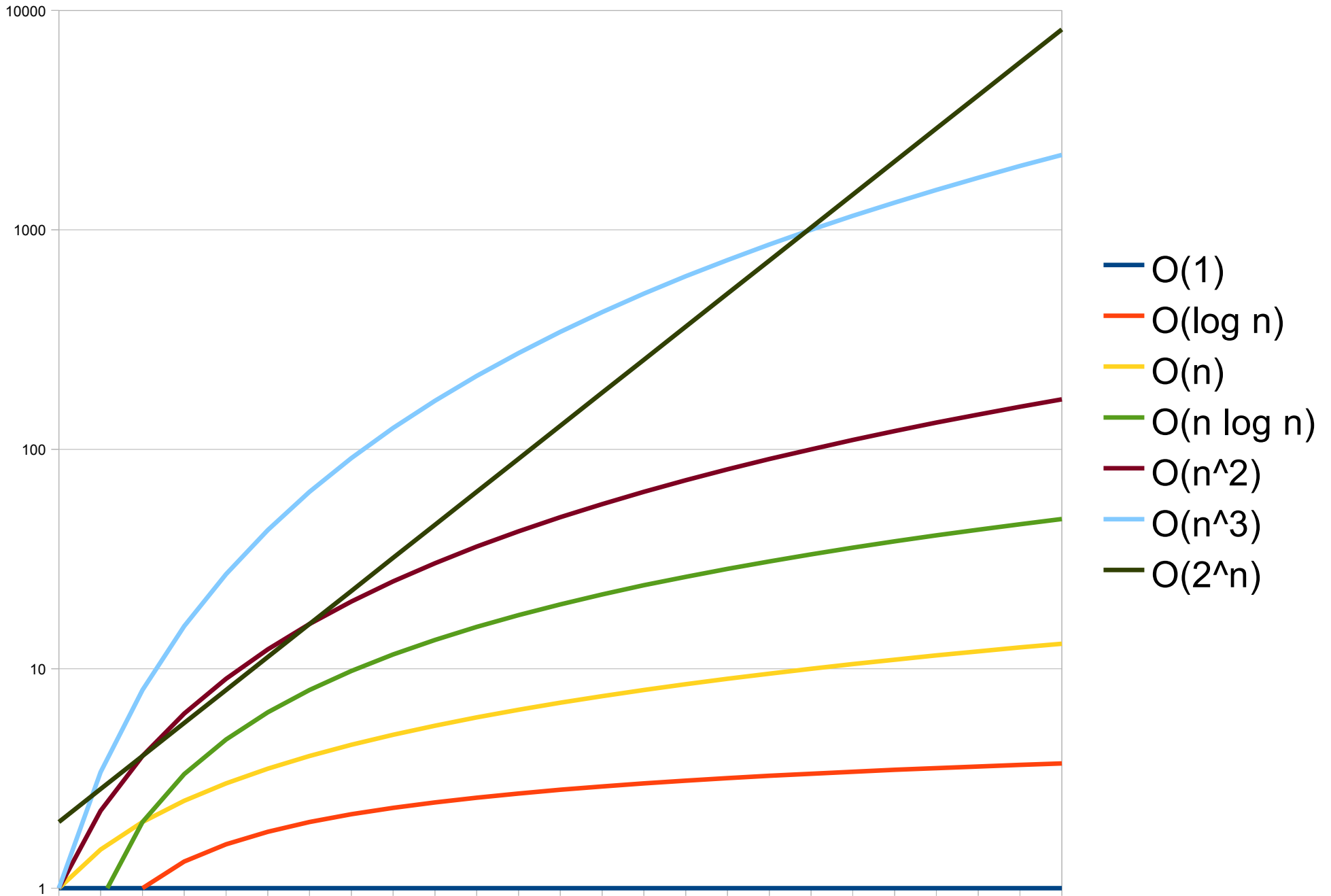
Growth Rates, Part Three



To Give You A Better Sense...



Once More with Logarithms



Comparison of Runtimes

(1 operation = 1 second)

Size	1	lg n	n	n log n	n ²	n ³	2 ⁿ
5	0:01	0:02	0:05	0:12	0:25	2:05	0:32
10	0:01	0:03	0:10	0:33	1:40	16:40	17:04
15	0:01	0:04	0:15	0:59	3:45	56:15	~9:00:00
20	0:01	0:04	0:20	1:26	6:40	2:13:20	~12 days
25	0:01	0:05	0:25	1:54	10:25	4:20:25	~1 year
30	0:01	0:05	0:30	2:27	15:00	7:30:00	~34 years
35	0:01	0:05	0:35	3:00	20:25	~12 hrs	~1000 years
40	0:01	0:05	0:40	3:33	26:40	~18 hrs	~35,000 yrs
45	0:01	0:05	0:45	4:07	33:45	~1 day	~1 million yrs
50	0:01	0:06	0:50	4:42	41:40	~1.5 days	~ 35 millon yr
55	0:01	0:06	0:55	5:18	50:25	~2 days	~1 billion yr
60	0:01	0:06	1:00	5:54	1:00:00	~2.5 days	~35 billion yr
65	0:01	0:06	1:05	6:31	1:10:25	~ 3 days	~ 1 trillion yr
70	0:01	0:06	1:10	7:09	1:20:40	~4 days	~ 35 trillion yr

Composing Functions

- How do you analyze a function that calls another function?
- Compute the big-O of the function being called and treat it as a black box.

Composition Example

Composition Example

```
void HideLetters(string& str)
{
    for (int i = 0; i < str.length(); ++i)
        if (IsAlpha(str[i]))
            str[i] = '?';
}
```

Composition Example

```
void HideLetters(string& str)
{
    for (int i = 0; i < str.length(); ++i)
        if (IsAlpha(str[i]))
            str[i] = '?';
}
```

Composition Example

```
void HideLetters(string& str)
{
    for (int i = 0; i < str.length(); ++i)
        if (IsAlpha(str[i]))
            str[i] = '?';
}
```

O(n)

Summary of Big-O

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

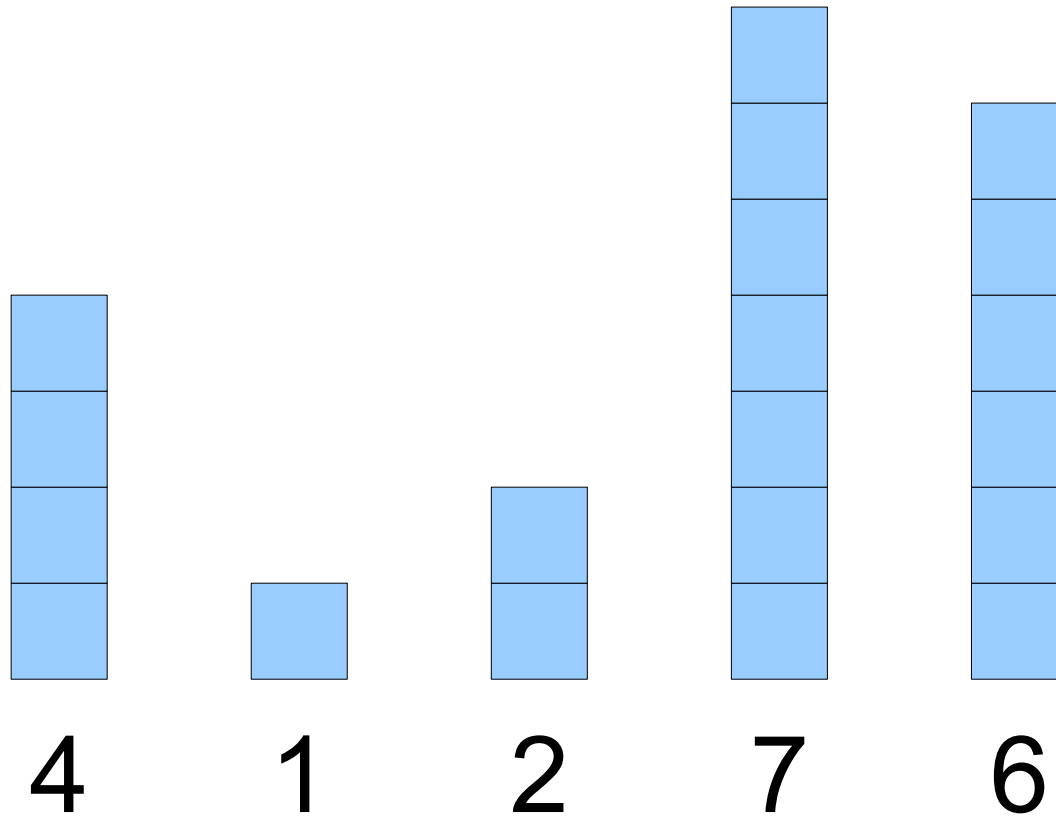
Sorting Algorithms

The Sorting Problem

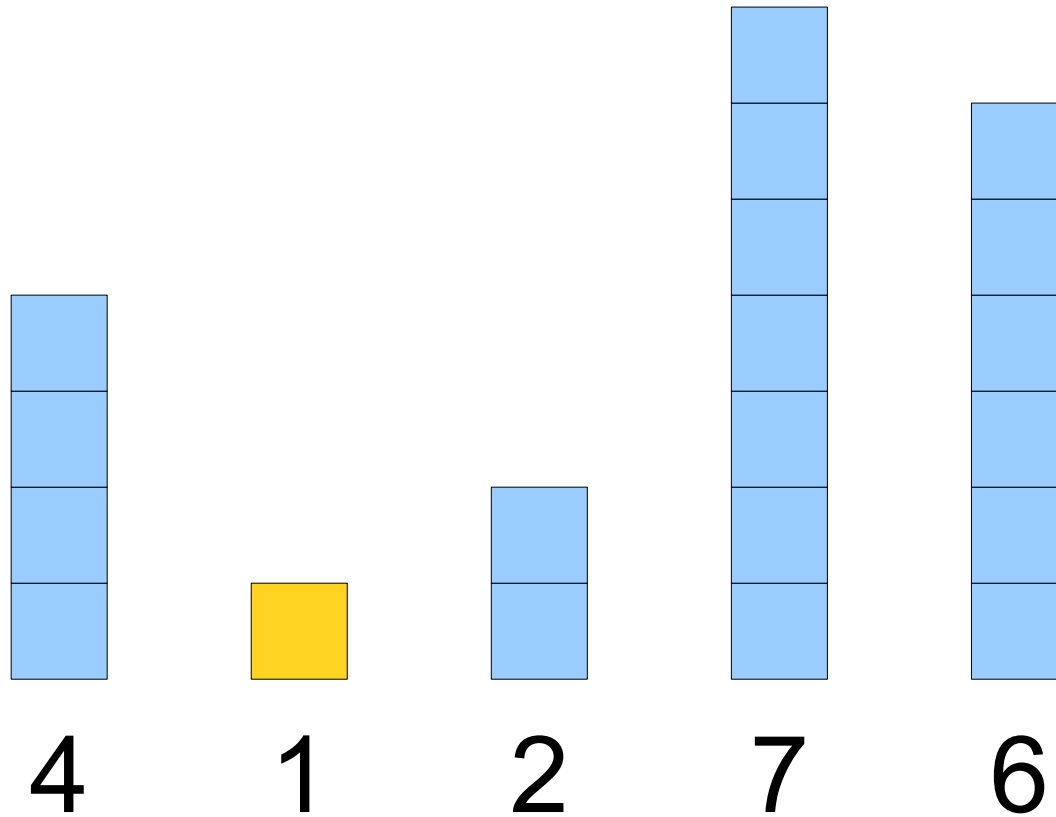
- Given a list of elements, sort those elements in ascending order.
- There are **many** ways to solve this problem.
- What is the **best** way to solve this problem?
- We'll use big-O to find out!

An Initial Idea: **Selection Sort**

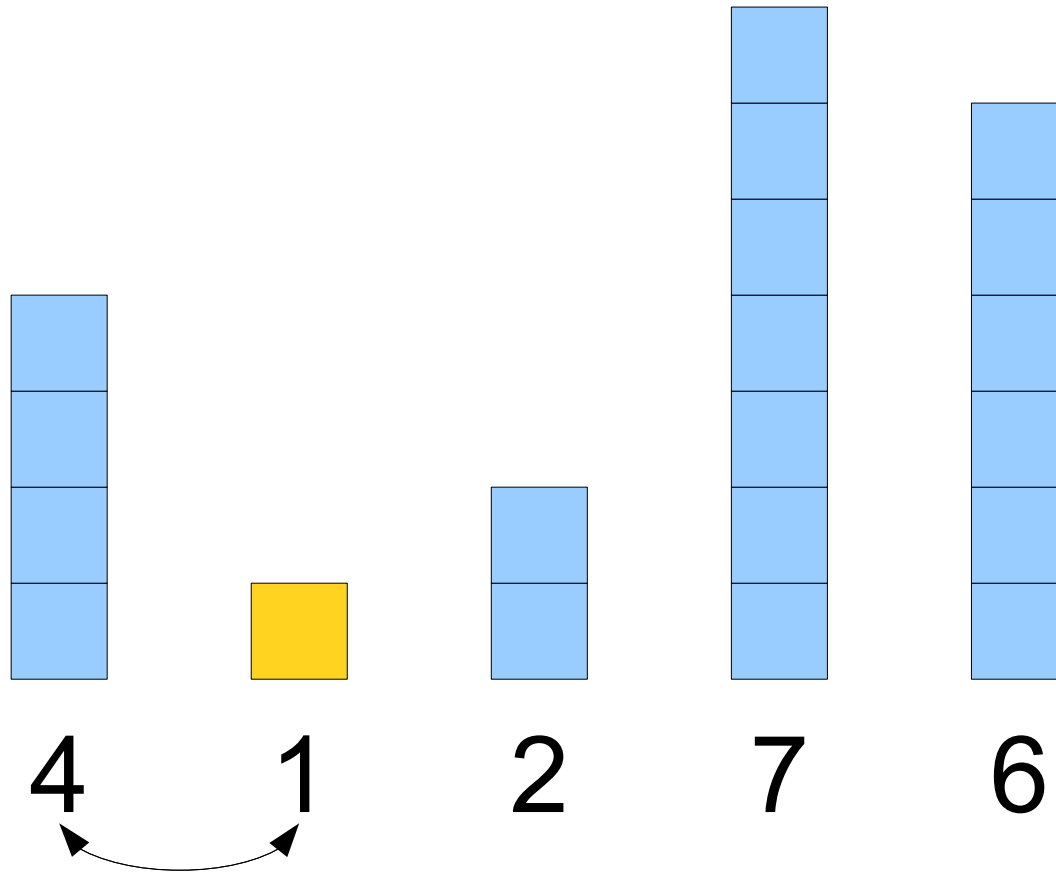
An Initial Idea: **Selection Sort**



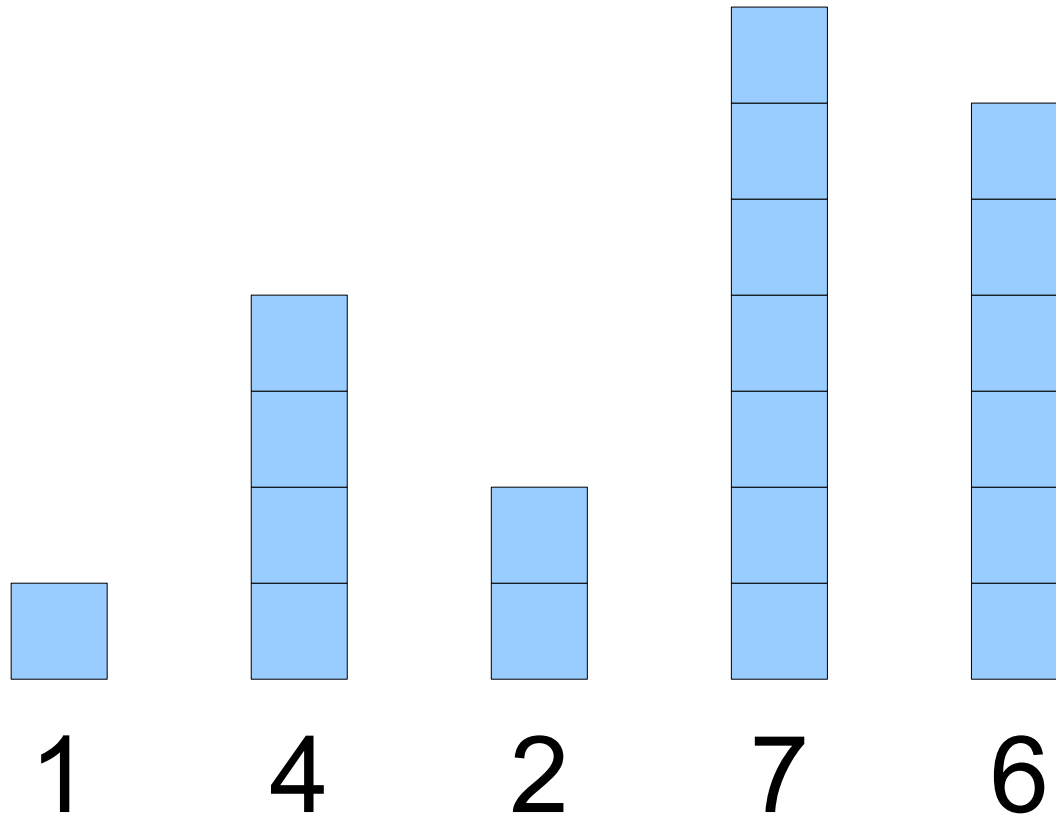
An Initial Idea: **Selection Sort**



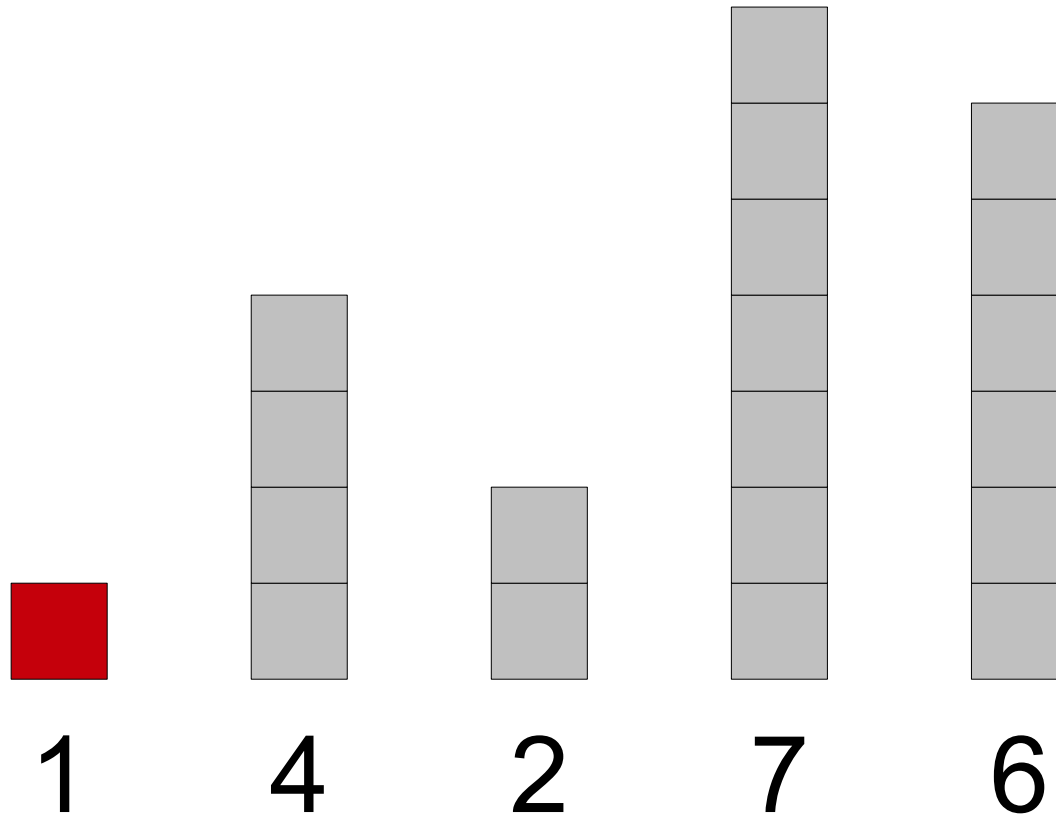
An Initial Idea: **Selection Sort**



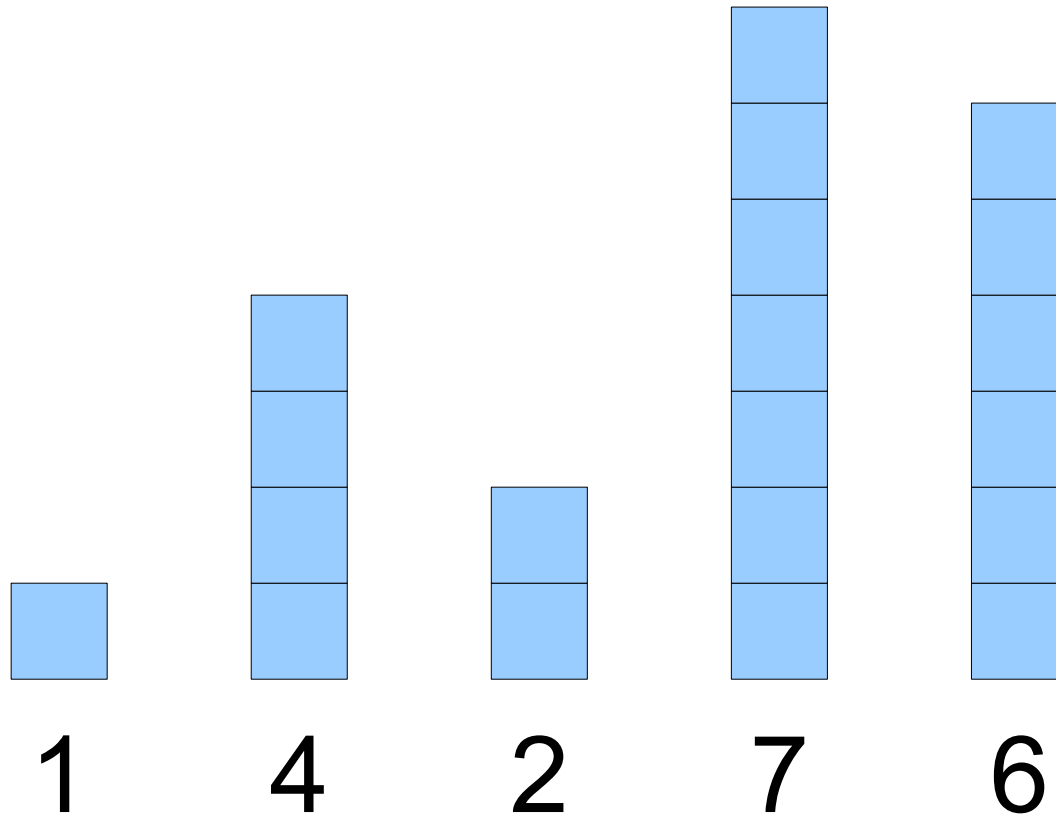
An Initial Idea: **Selection Sort**



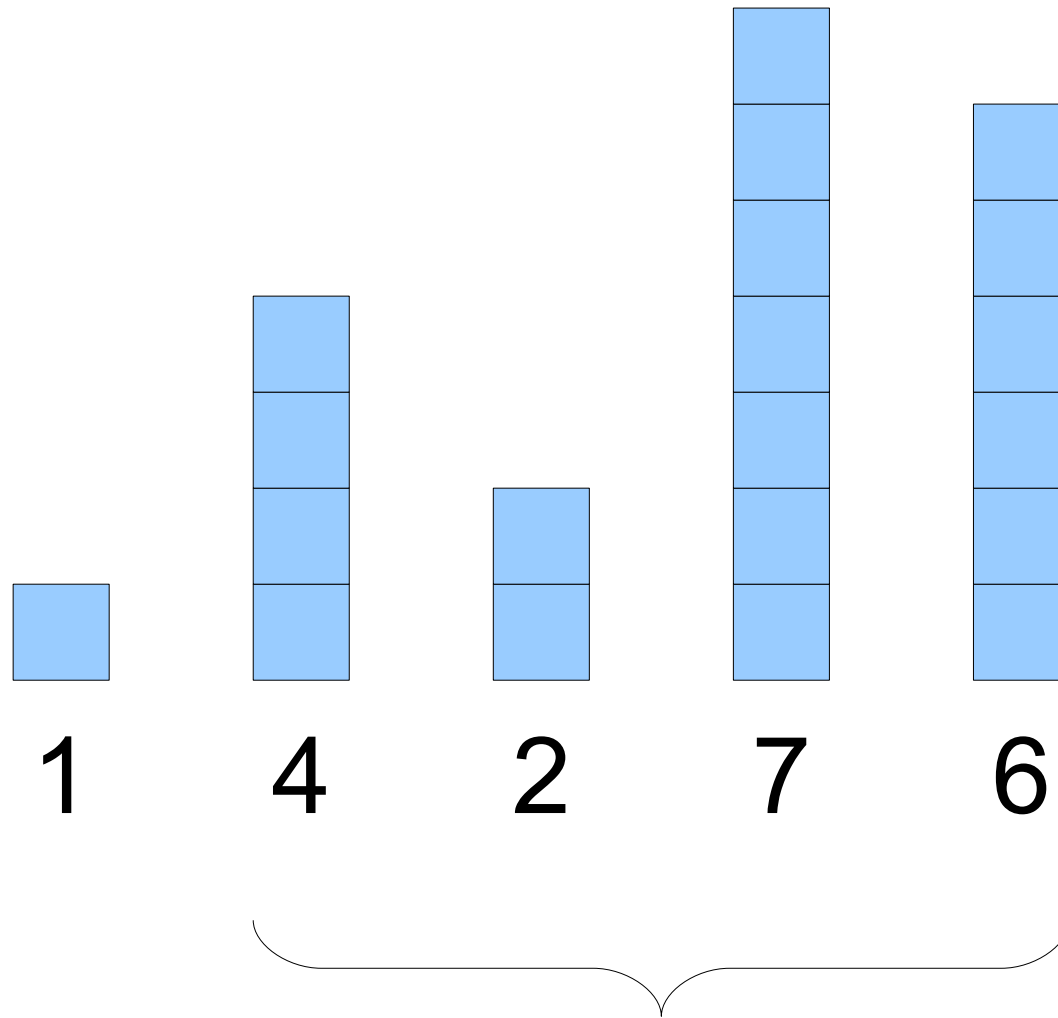
An Initial Idea: **Selection Sort**



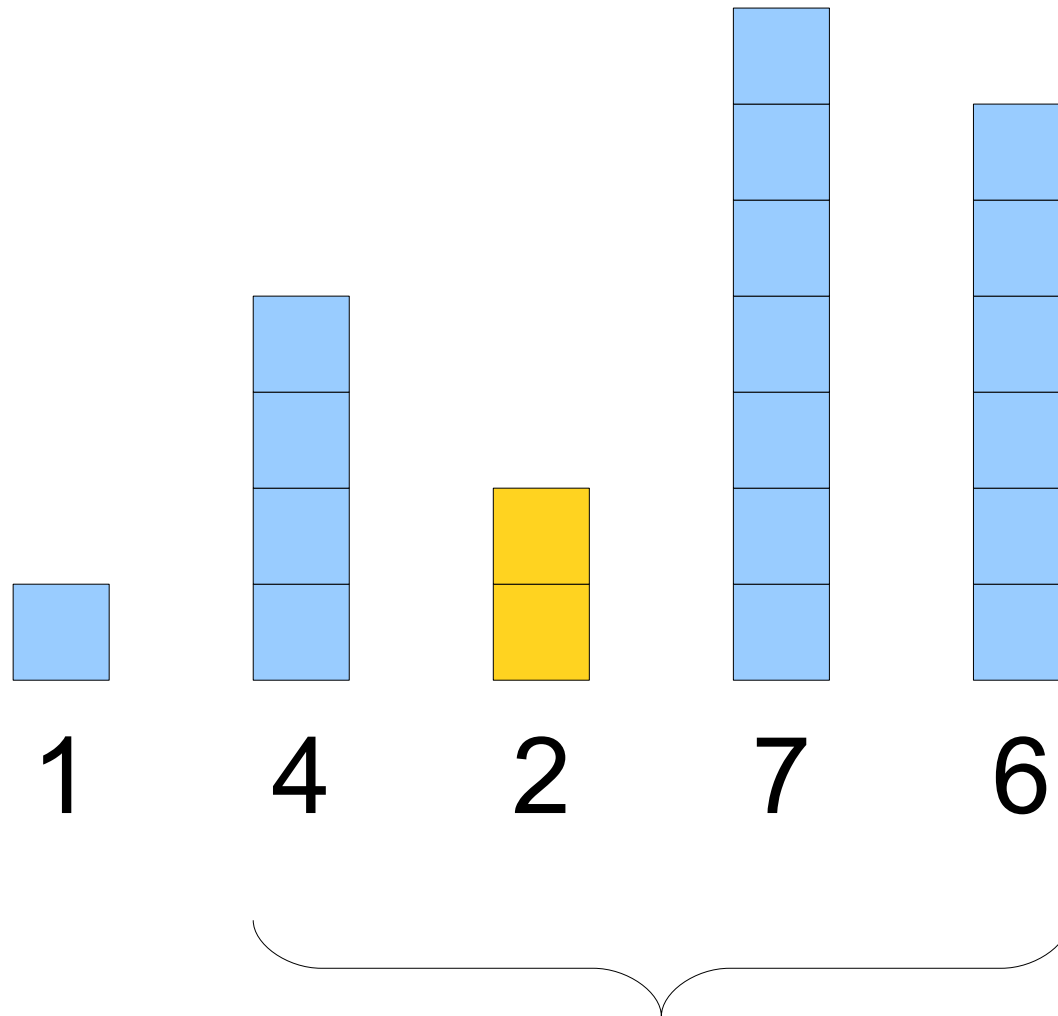
An Initial Idea: **Selection Sort**



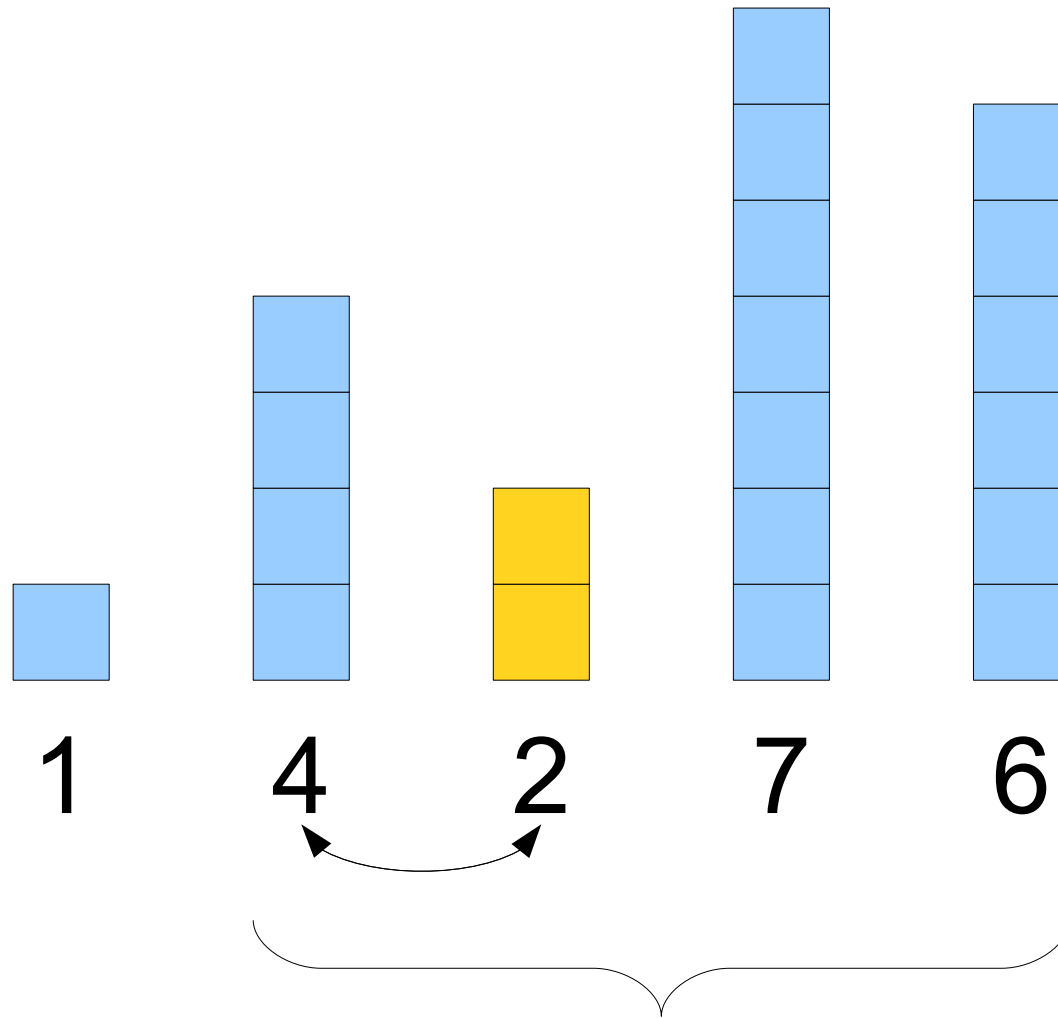
An Initial Idea: **Selection Sort**



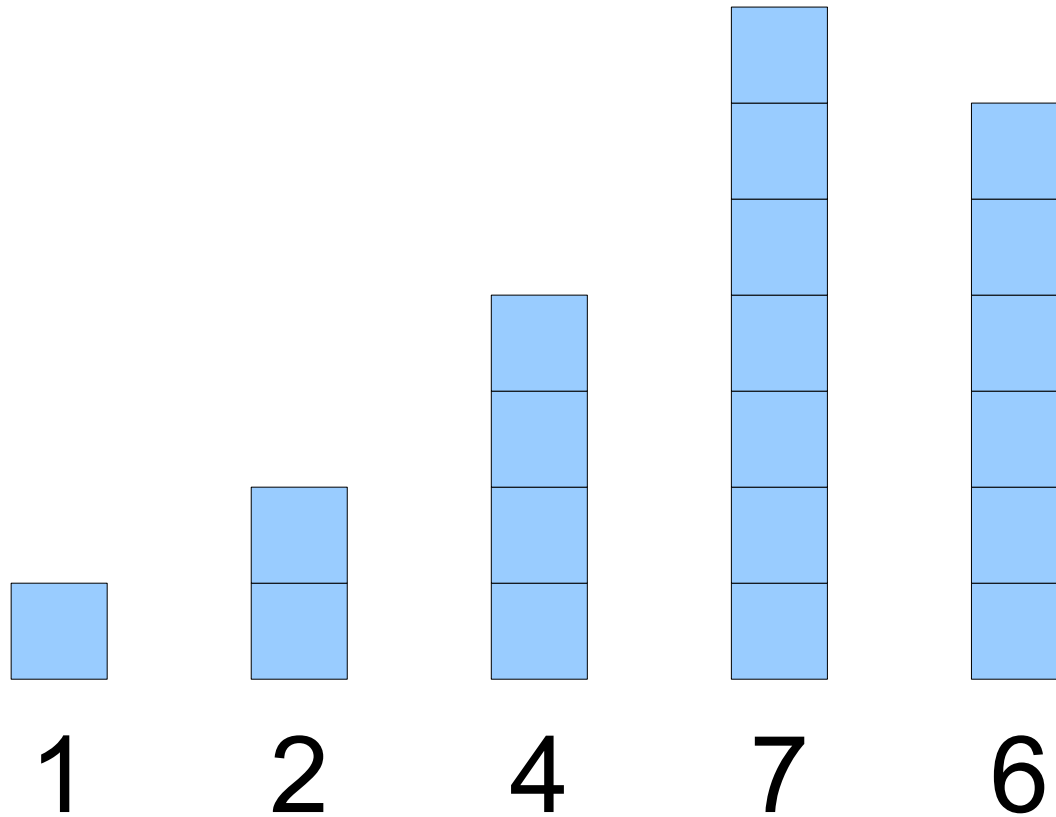
An Initial Idea: **Selection Sort**



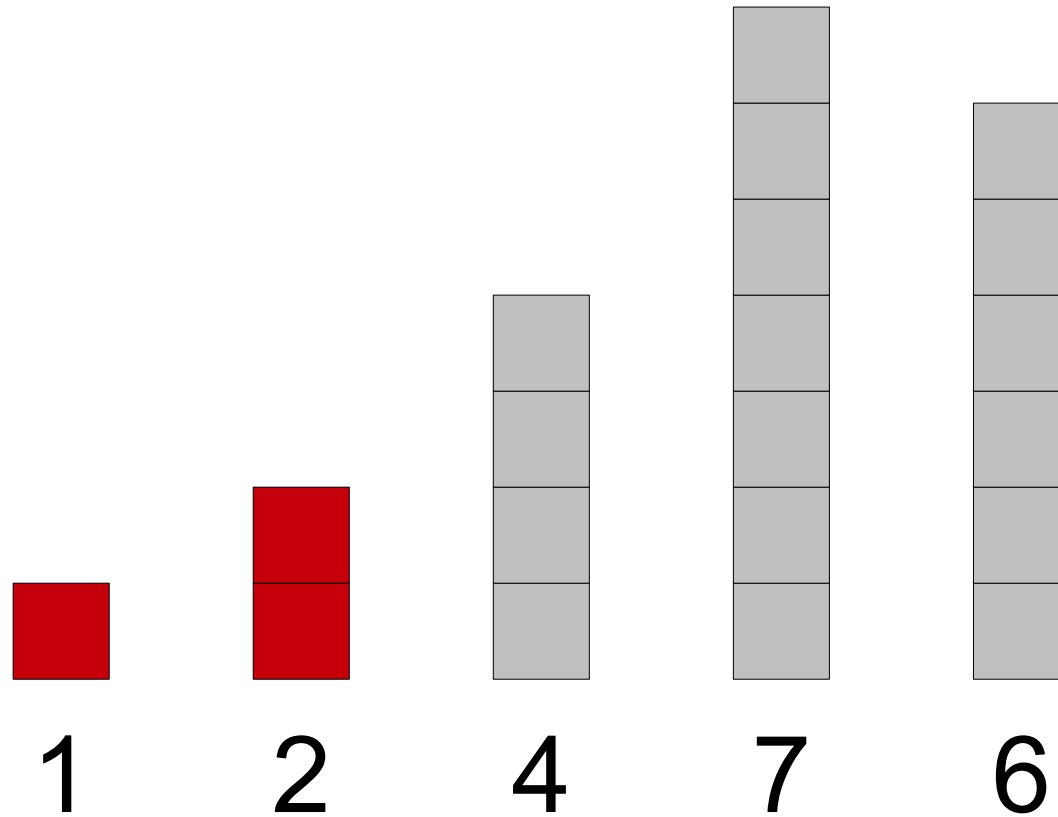
An Initial Idea: Selection Sort



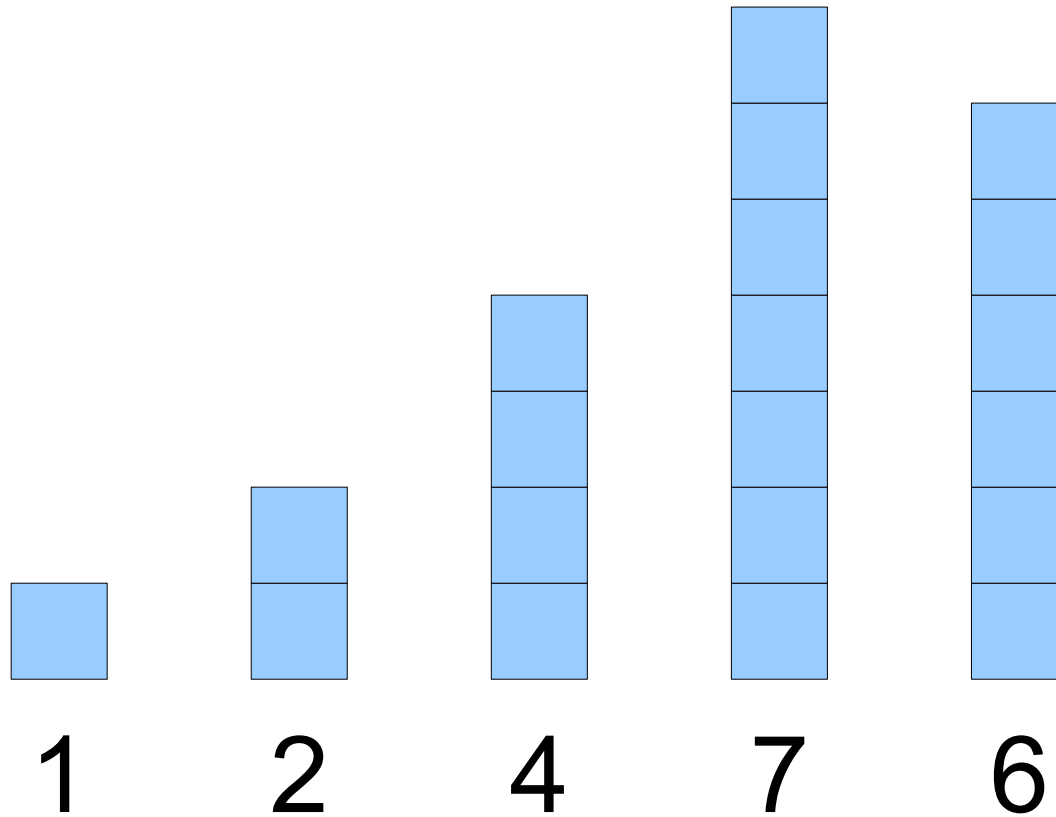
An Initial Idea: **Selection Sort**



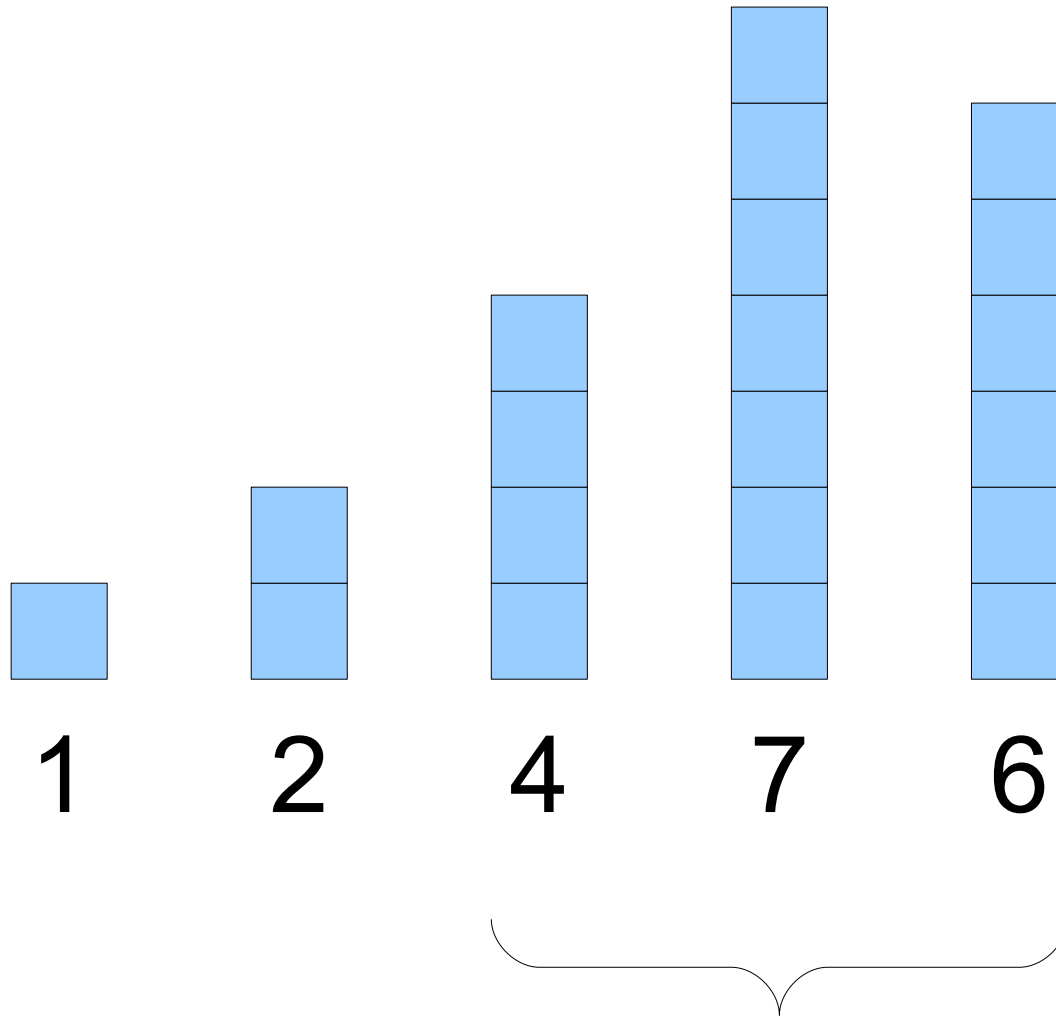
An Initial Idea: **Selection Sort**



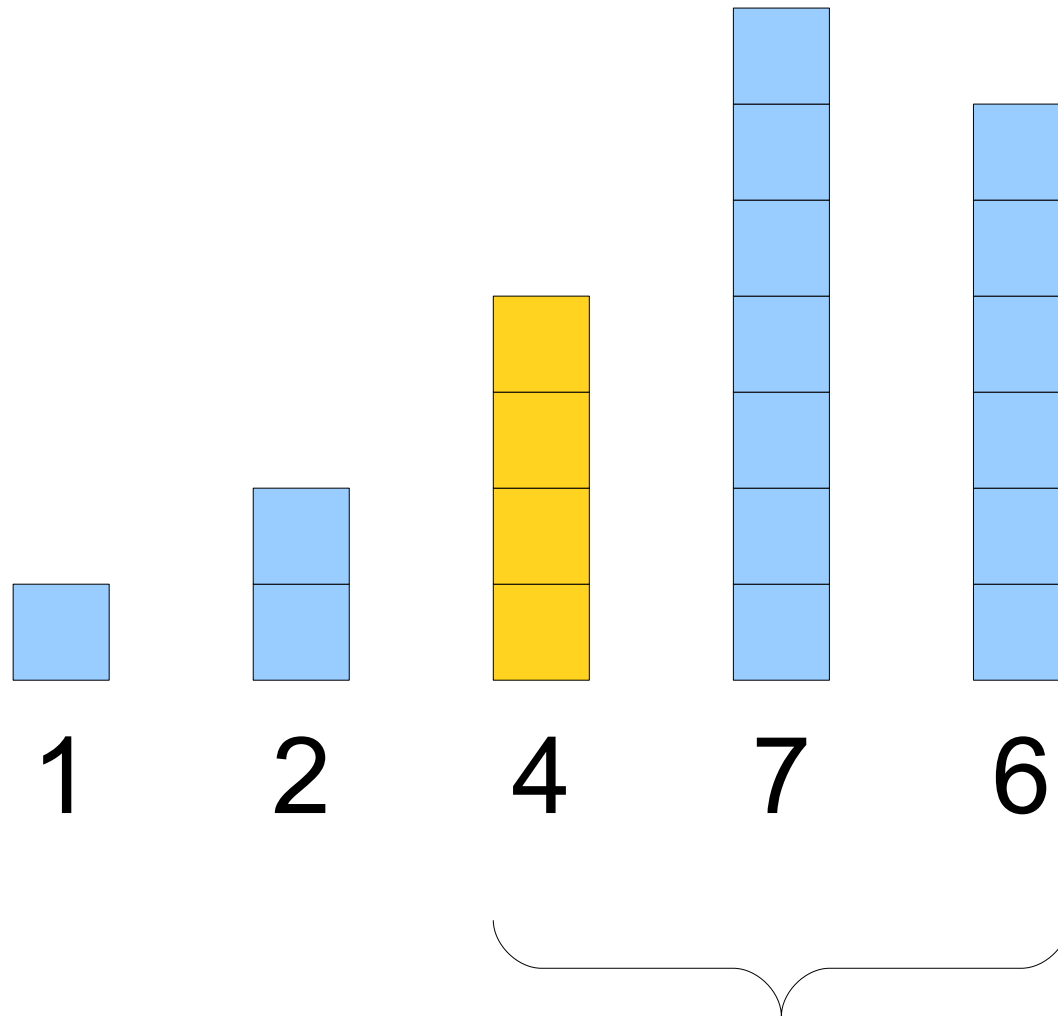
An Initial Idea: **Selection Sort**



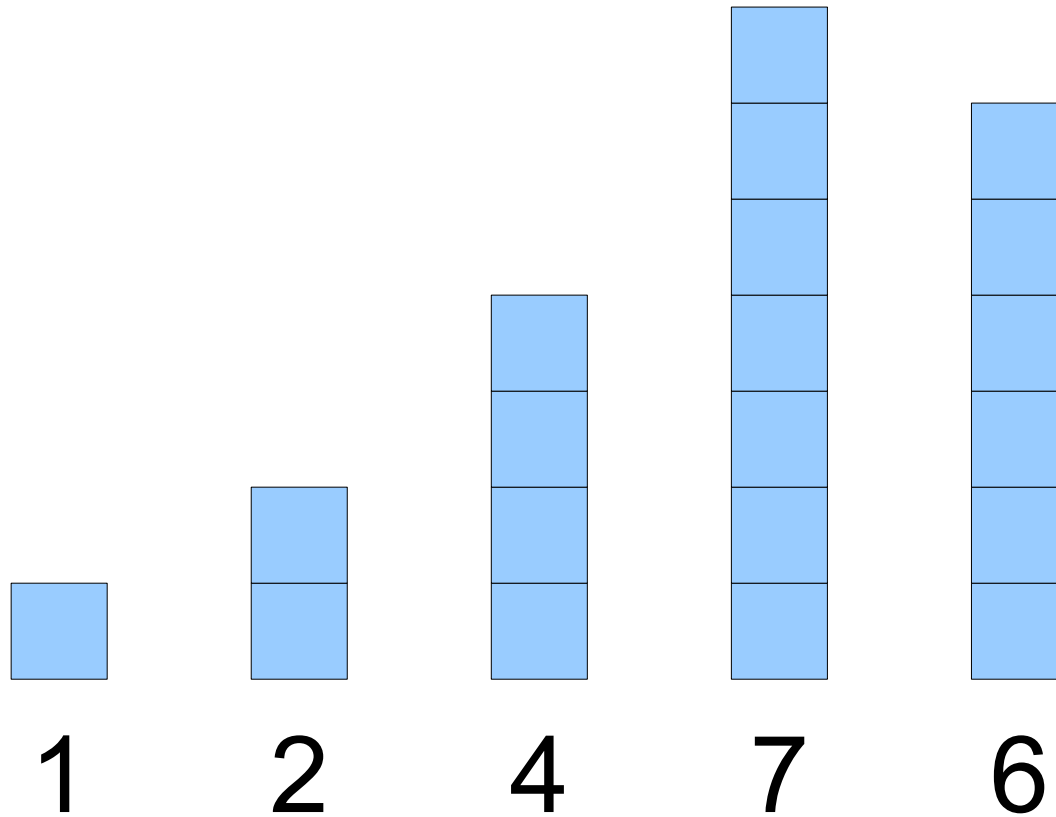
An Initial Idea: **Selection Sort**



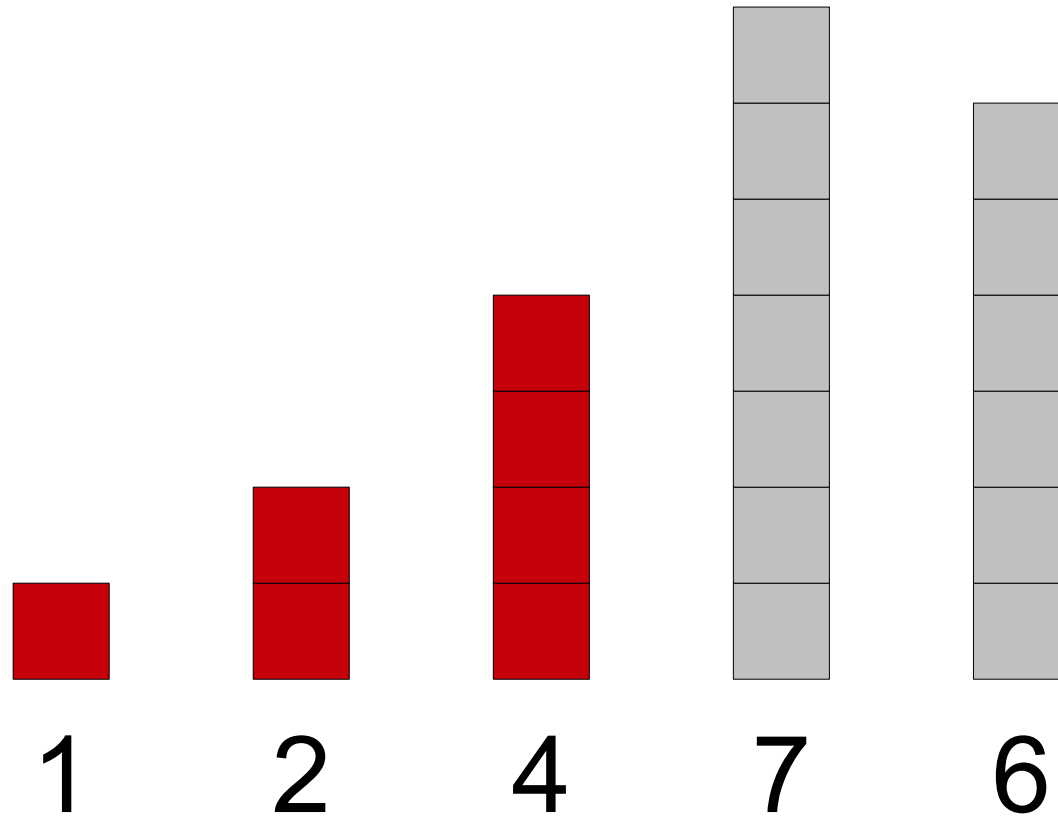
An Initial Idea: Selection Sort



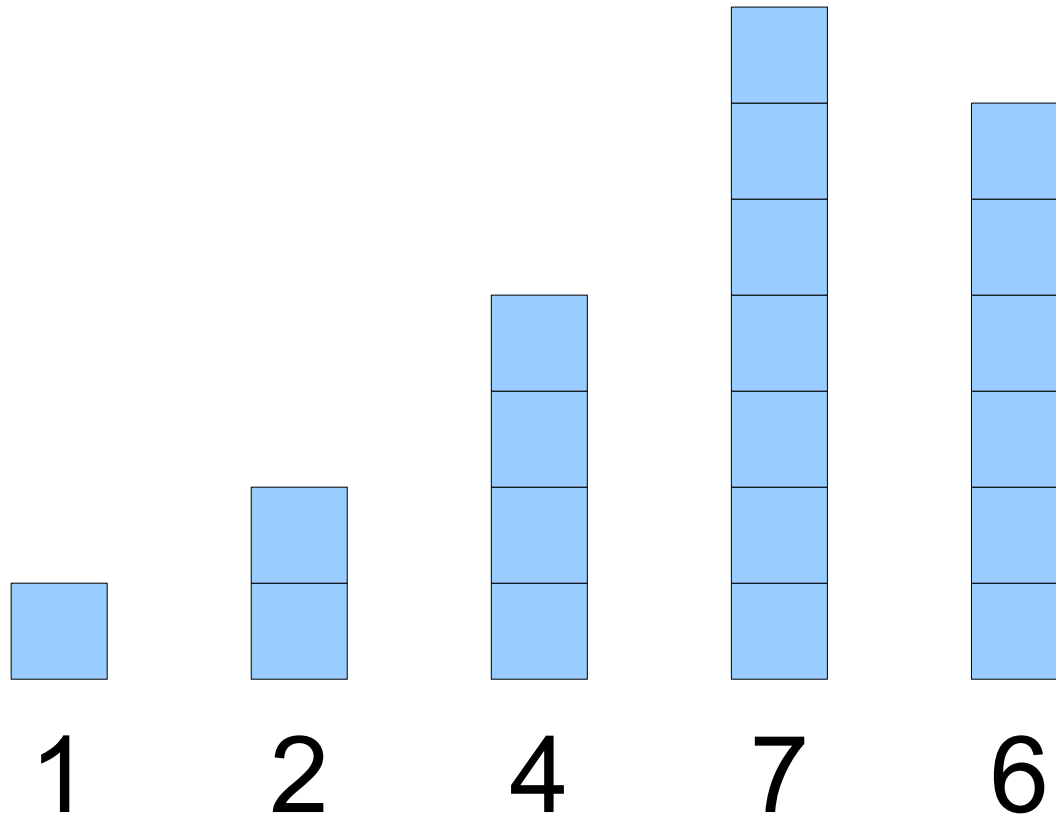
An Initial Idea: **Selection Sort**



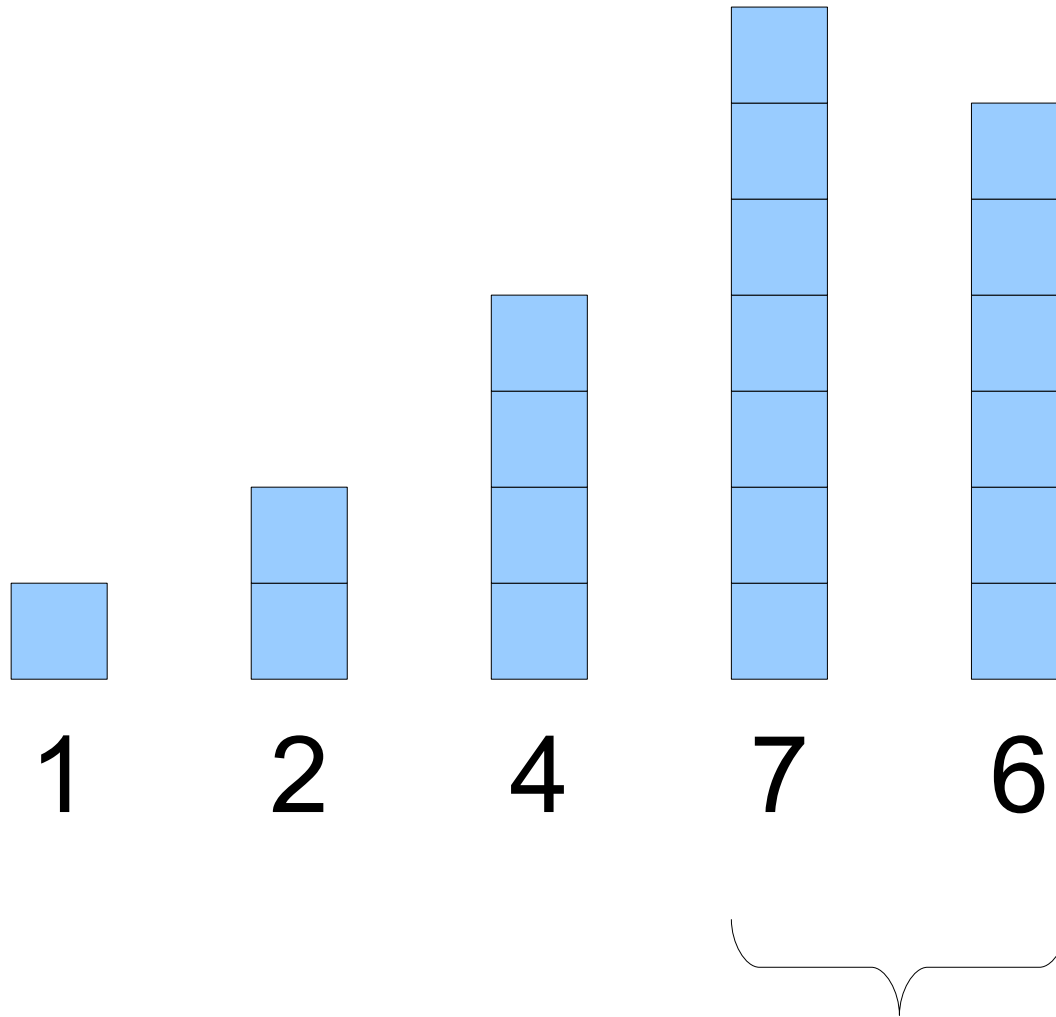
An Initial Idea: **Selection Sort**



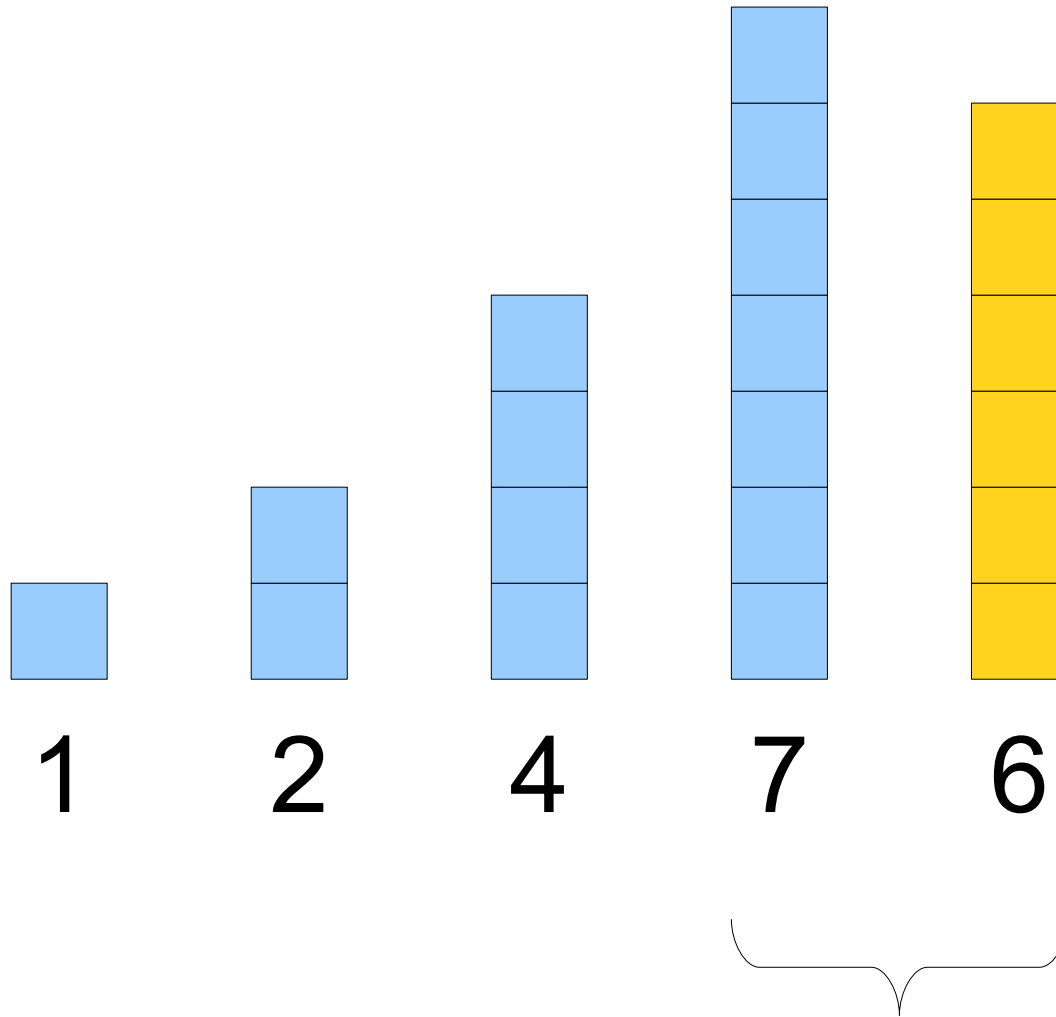
An Initial Idea: **Selection Sort**



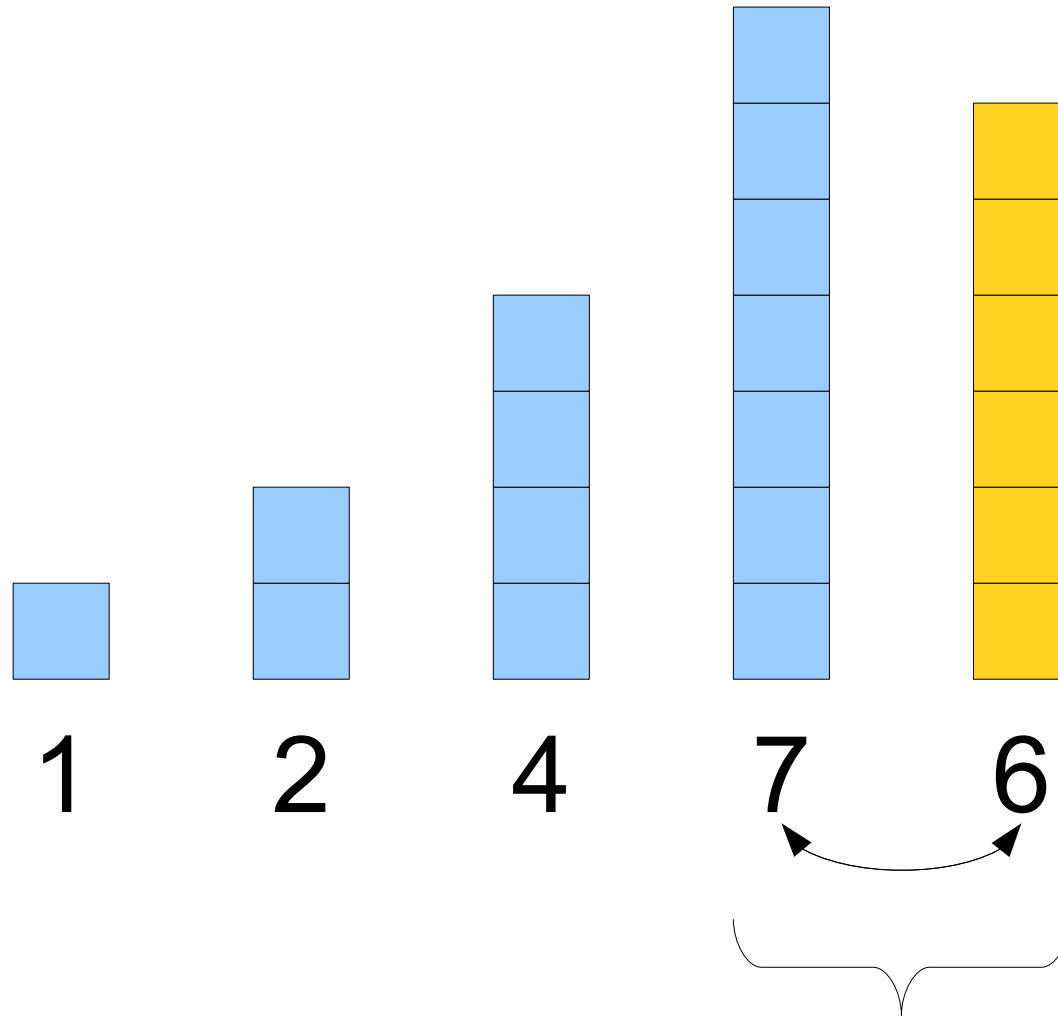
An Initial Idea: **Selection Sort**



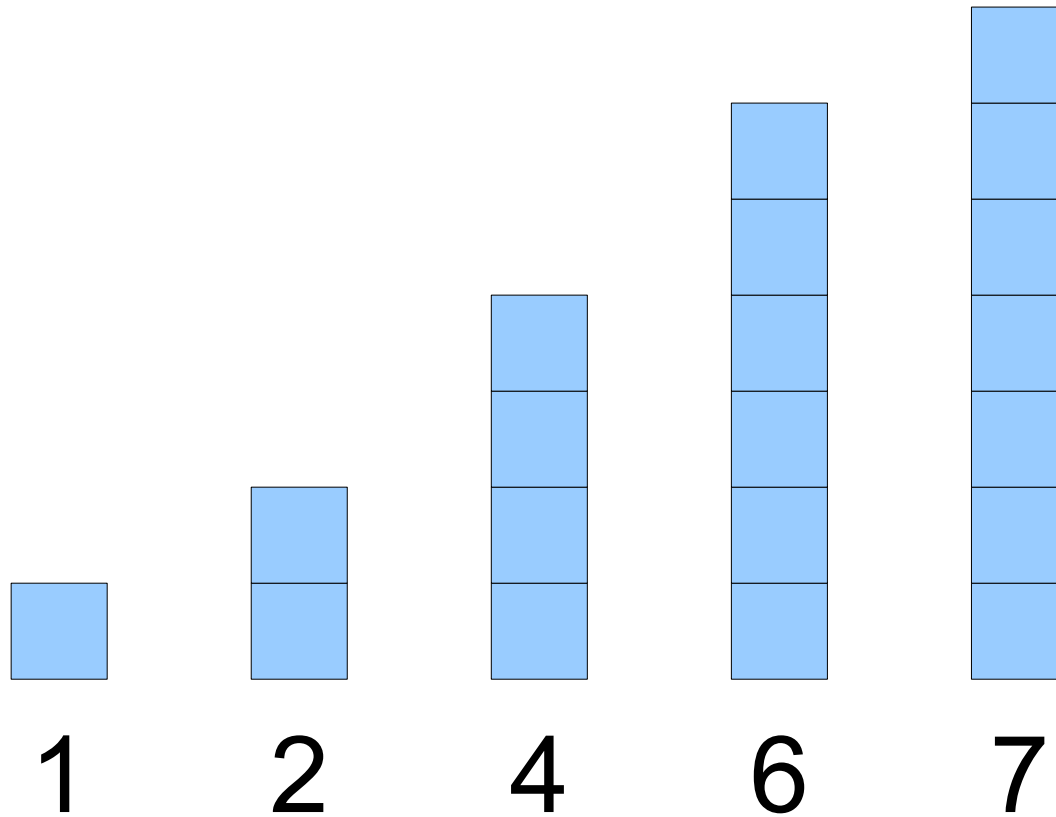
An Initial Idea: Selection Sort



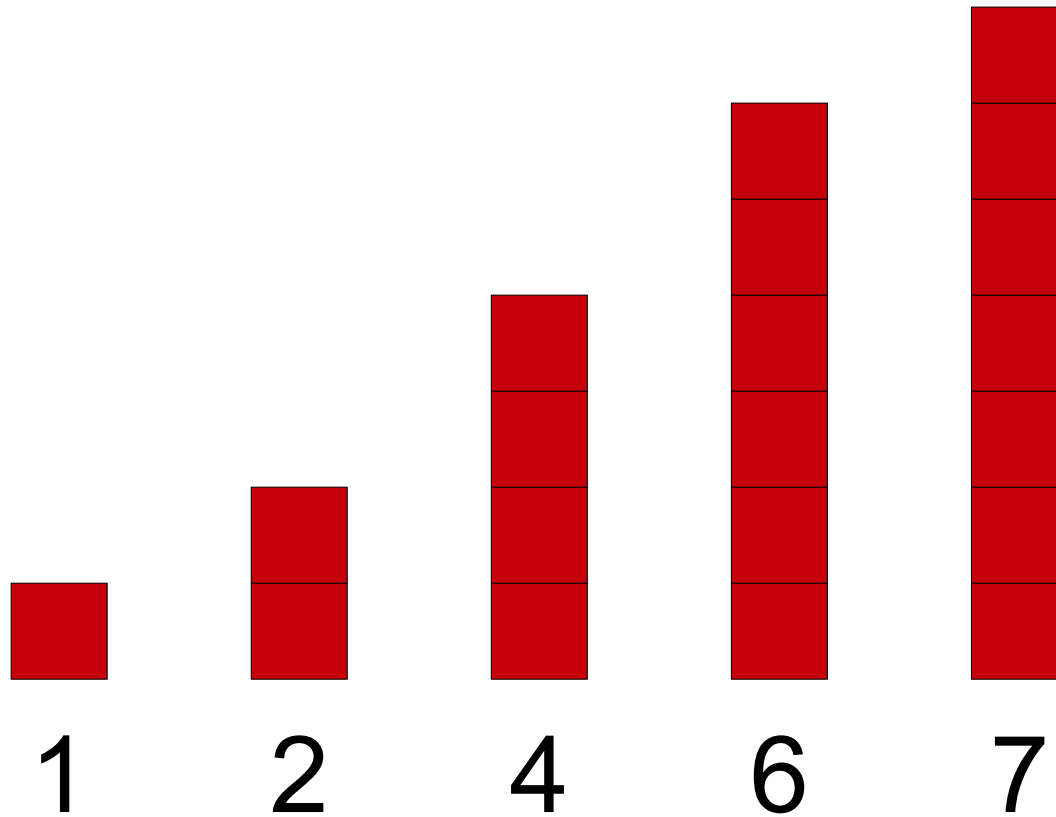
An Initial Idea: **Selection Sort**



An Initial Idea: **Selection Sort**



An Initial Idea: **Selection Sort**



Selection Sort in Code

Selection Sort in Code

```
void SelectionSort (Vector<int>& v)
{
    for (int i = 0; i < v.size(); i++)
    {
        int minIndex = MinElementIndex(v, i);
        swap(v[i], v[minIndex]);
    }
}

int MinElementIndex(Vector<int>& v, int startIndex)
{
    int bestIndex = startIndex;

    for (int i = startIndex + 1; i < v.size(); i++)
        if (v[i] < v[bestIndex])
            bestIndex = i;

    return bestIndex;
}
```

Selection Sort in Code

```
void SelectionSort (Vector<int>& v)
```

```
{  
    for (int i = 0; i < v.size(); i++)  
    {  
        int minIndex = MinElementIndex(v, i);  
        swap(v[i], v[minIndex]);  
    }  
}
```

```
int MinElementIndex(Vector<int>& v, int startIndex)
```

```
{  
    int bestIndex = startIndex;  
  
    for (int i = startIndex + 1; i < v.size(); i++)  
        if (v[i] < v[bestIndex])  
            bestIndex = i;
```

```
    return bestIndex;
```

```
}
```

n - i work

Selection Sort in Code

```
void SelectionSort (Vector<int>& v)
```

```
{  
    for (int i = 0; i < v.size(); i++)  
    {  
        int minIndex = MinElementIndex(v, i);  
        swap(v[i], v[minIndex]);  
    }  
}
```

```
int MinElementIndex(Vector<int>& v, int startIndex)
```

```
{  
    int bestIndex = startIndex;  
  
    for (int i = startIndex + 1; i < v.size(); i++)  
        if (v[i] < v[bestIndex])  
            bestIndex = i;
```

```
    return bestIndex;
```

```
}
```

n - i work

Selection Sort in Code

```
void SelectionSort (Vector<int>& v)
```

```
{  
    for (int i = 0; i < v.size(); i++)  
    {  
        int minIndex = MinElementIndex(v, i);  
        swap(v[i], v[minIndex]);  
    }  
}
```

```
int MinElementIndex(Vector<int>& v, int startIndex)
```

```
{  
    int bestIndex = startIndex;  
  
    for (int i = startIndex + 1; i < v.size(); i++)  
        if (v[i] < v[bestIndex])  
            bestIndex = i;  
  
    return bestIndex;  
}
```

n - i work

Selection Sort in Code

```
void SelectionSort (Vector<int>& v)
```

```
{  
    for (int i = 0; i < v.size(); i++)  
    {  
        int minIndex = MinElementIndex(v, i);  
        swap(v[i], v[minIndex]);  
    }  
}
```

$O(n^2)$

```
int MinElementIndex(Vector<int>& v, int startIndex)
```

```
{  
    int bestIndex = startIndex;  
  
    for (int i = startIndex + 1; i < v.size(); i++)  
        if (v[i] < v[bestIndex])  
            bestIndex = i;
```

$n - i$ work

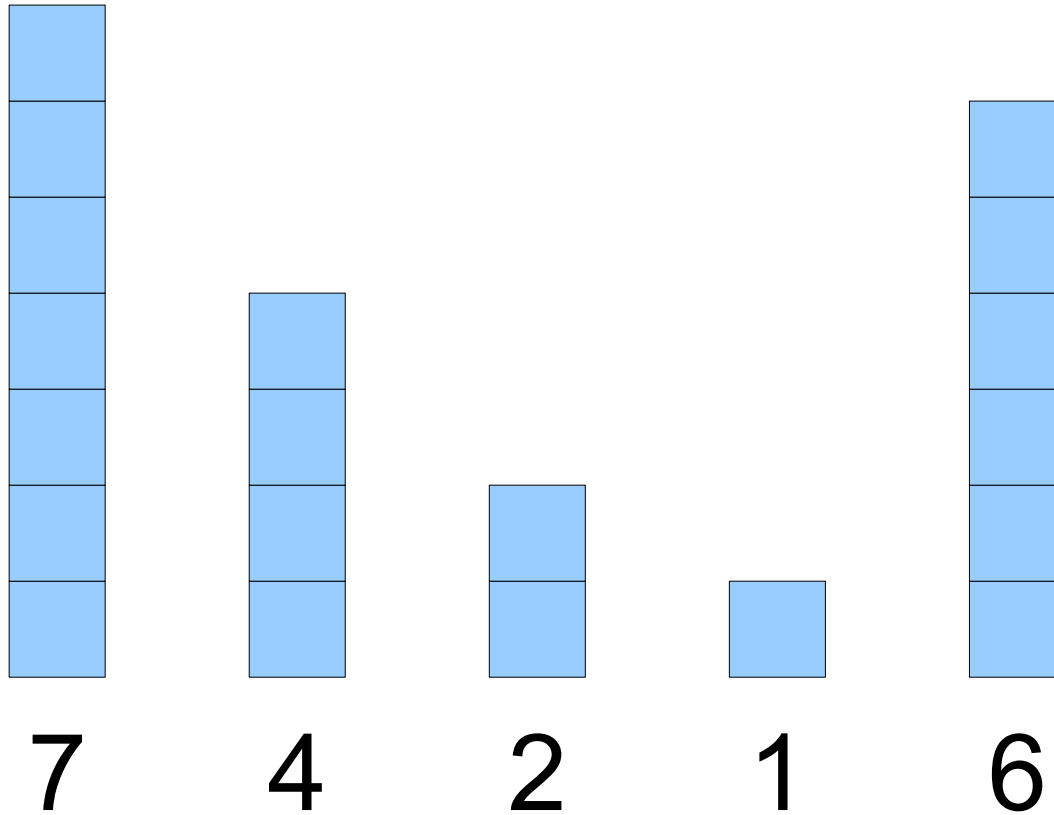
```
    return bestIndex;  
}
```

Notes on Selection Sort

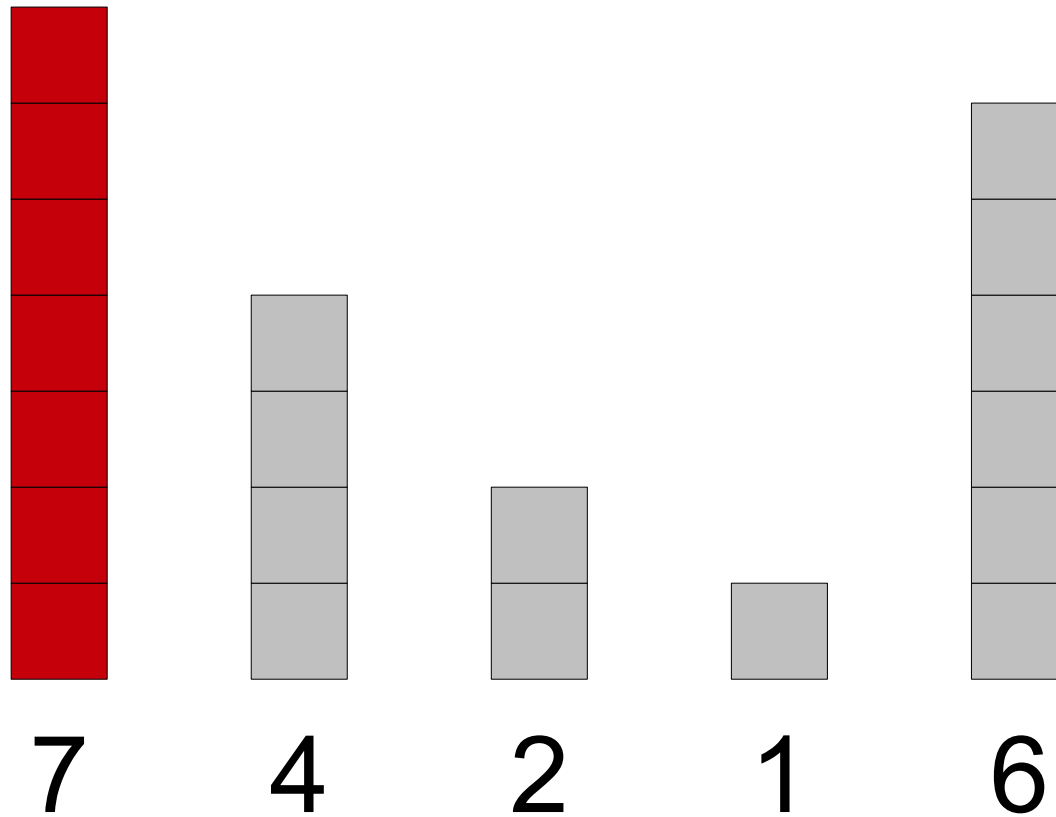
- Selection sort is $O(n^2)$ in the worst case.
- How about the best case?
- Also $O(n^2)$
- Selection sort *always* takes $O(n^2)$ time.
- Notation: Selection sort is $\Theta(n^2)$.

Another Idea: **Insertion Sort**

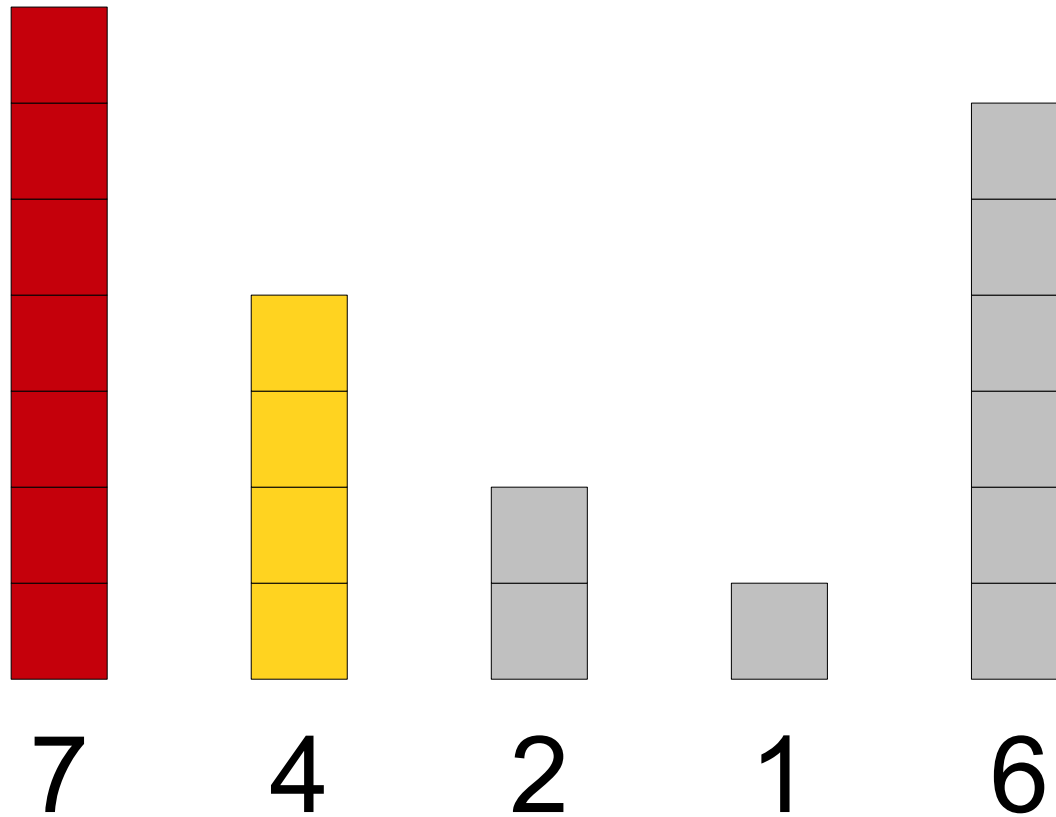
Another Idea: Insertion Sort



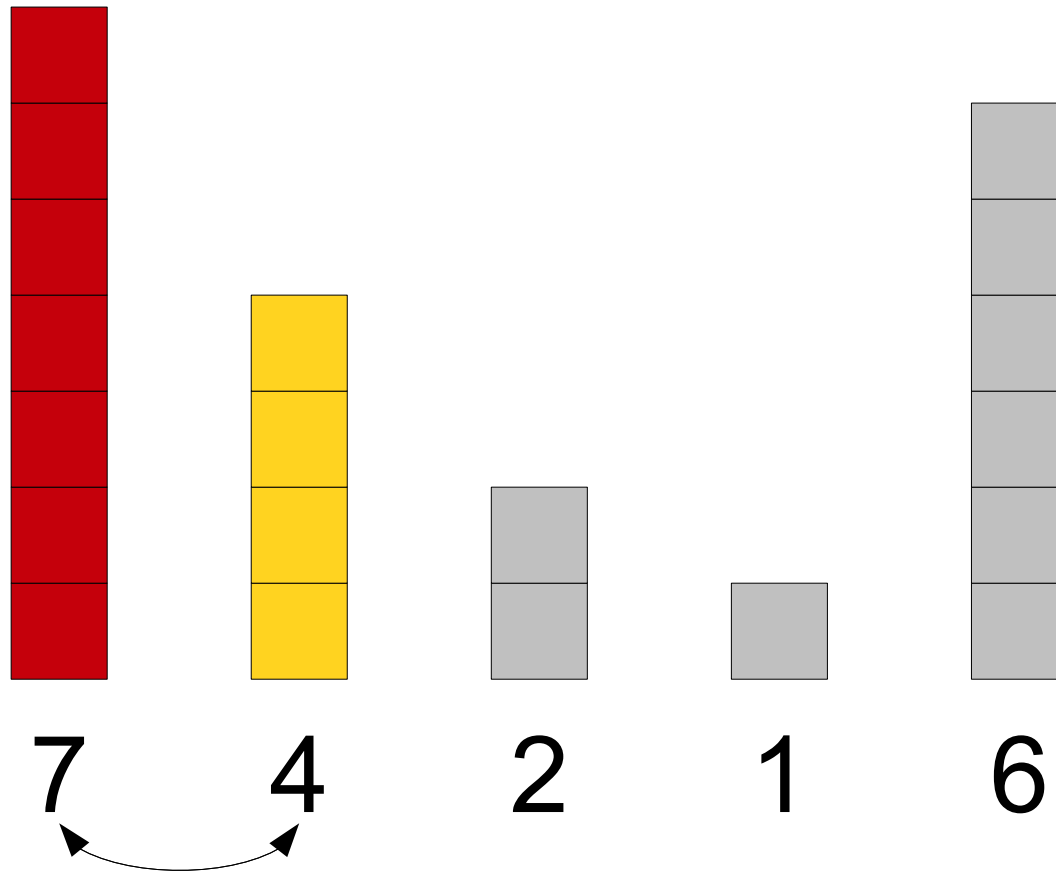
Another Idea: Insertion Sort



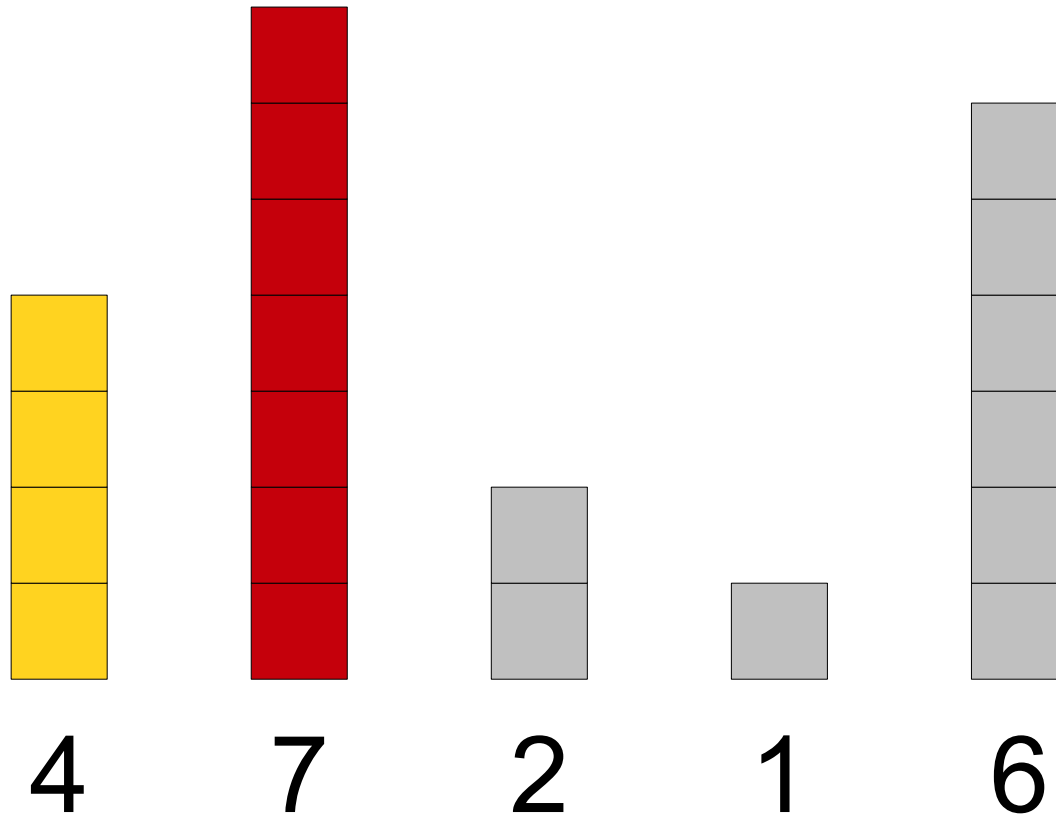
Another Idea: Insertion Sort



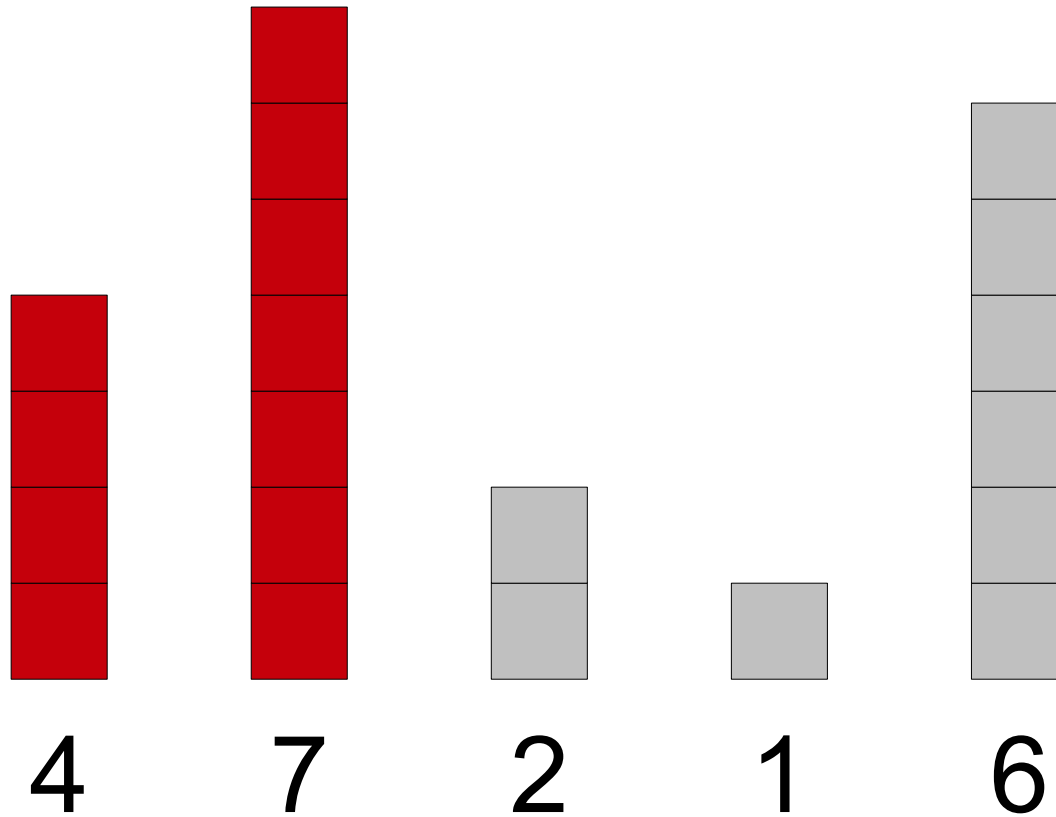
Another Idea: Insertion Sort



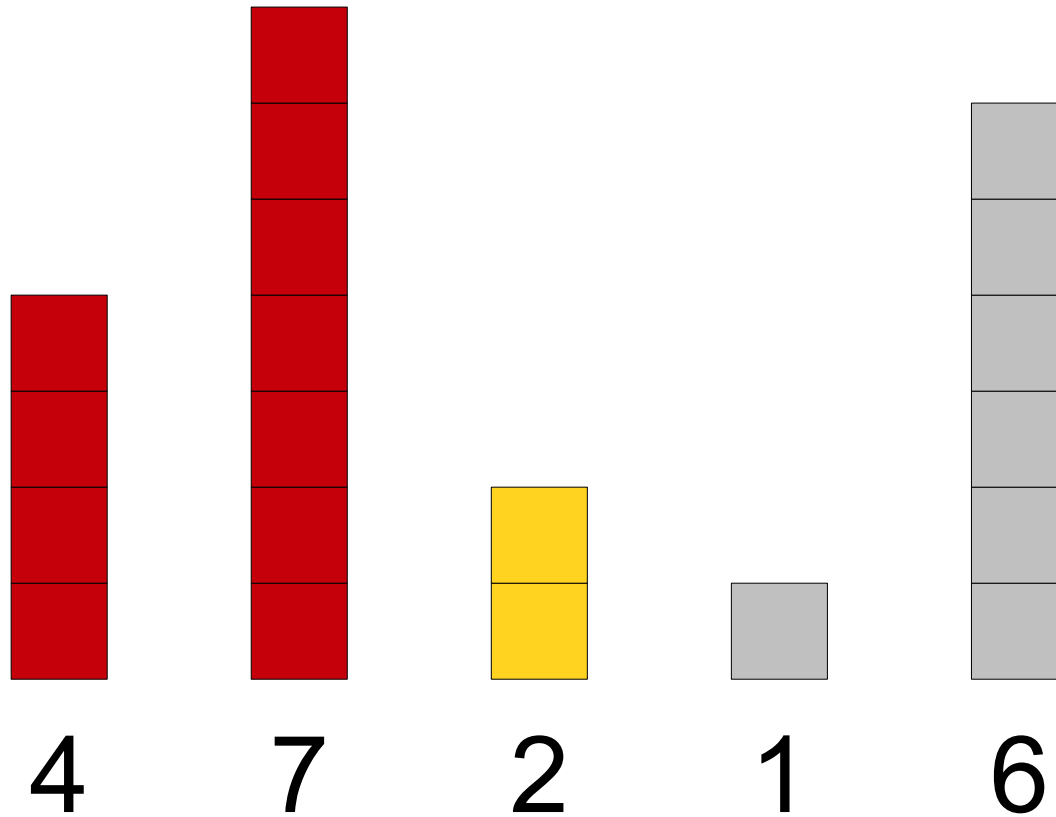
Another Idea: Insertion Sort



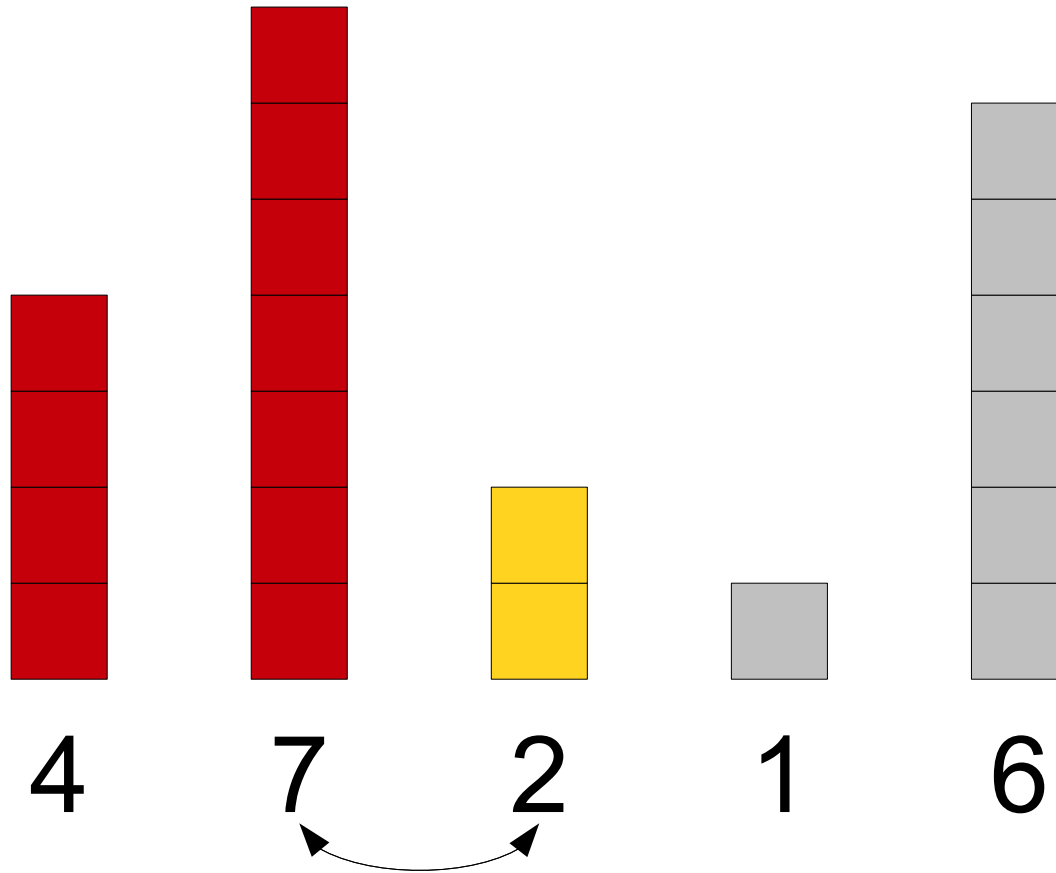
Another Idea: Insertion Sort



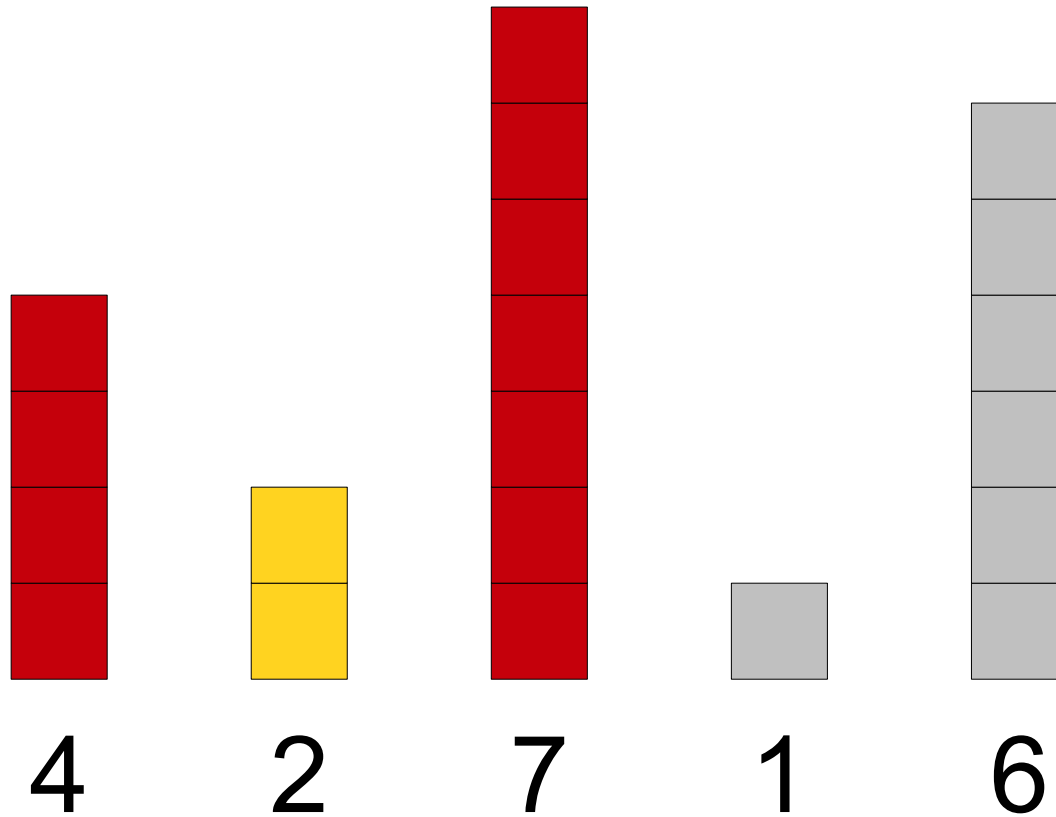
Another Idea: Insertion Sort



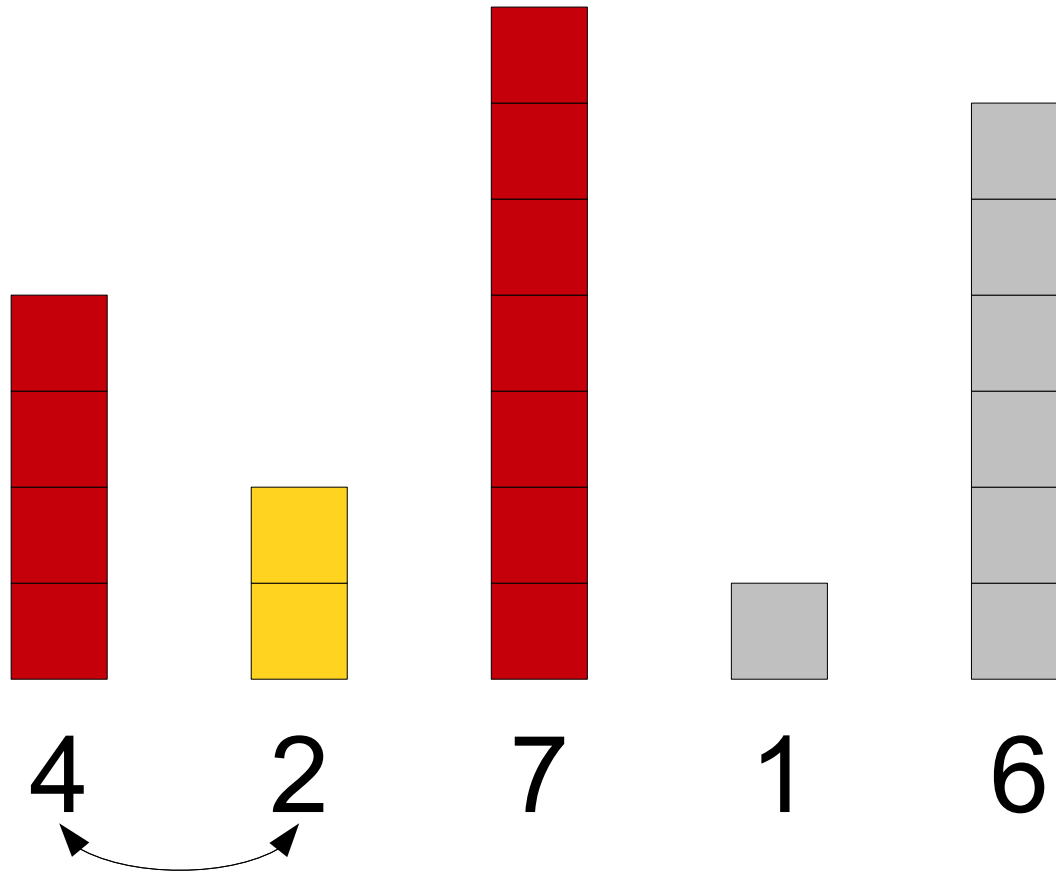
Another Idea: Insertion Sort



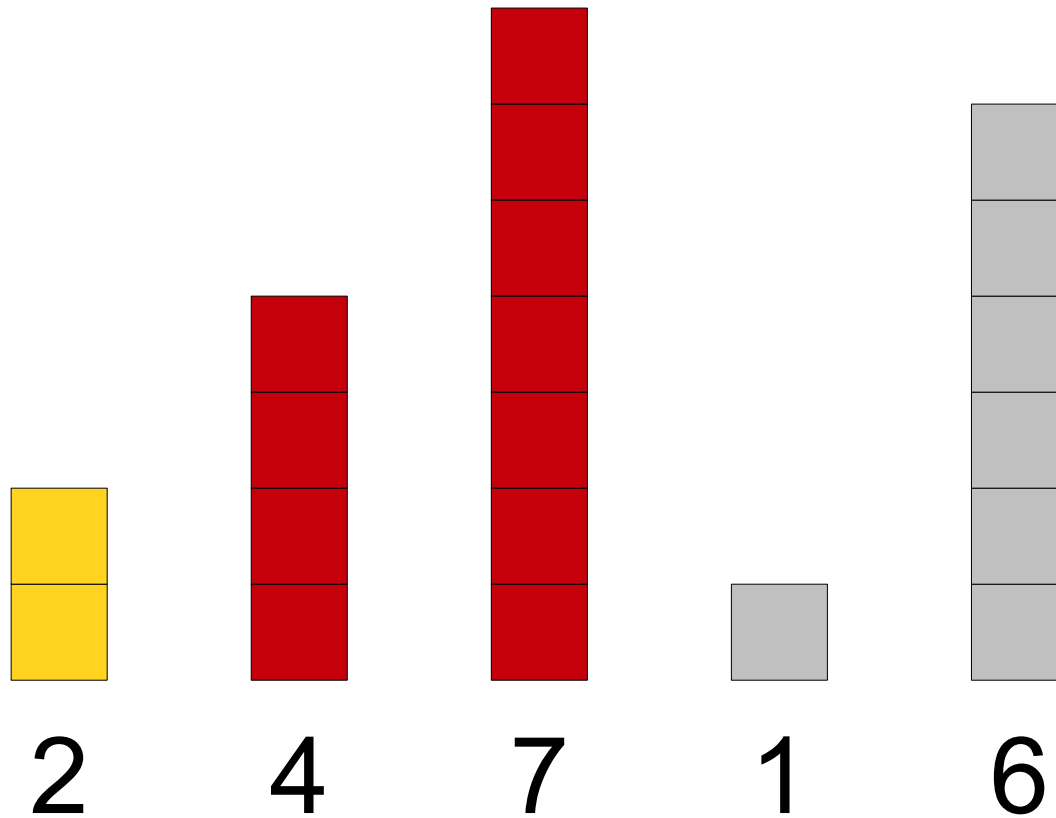
Another Idea: Insertion Sort



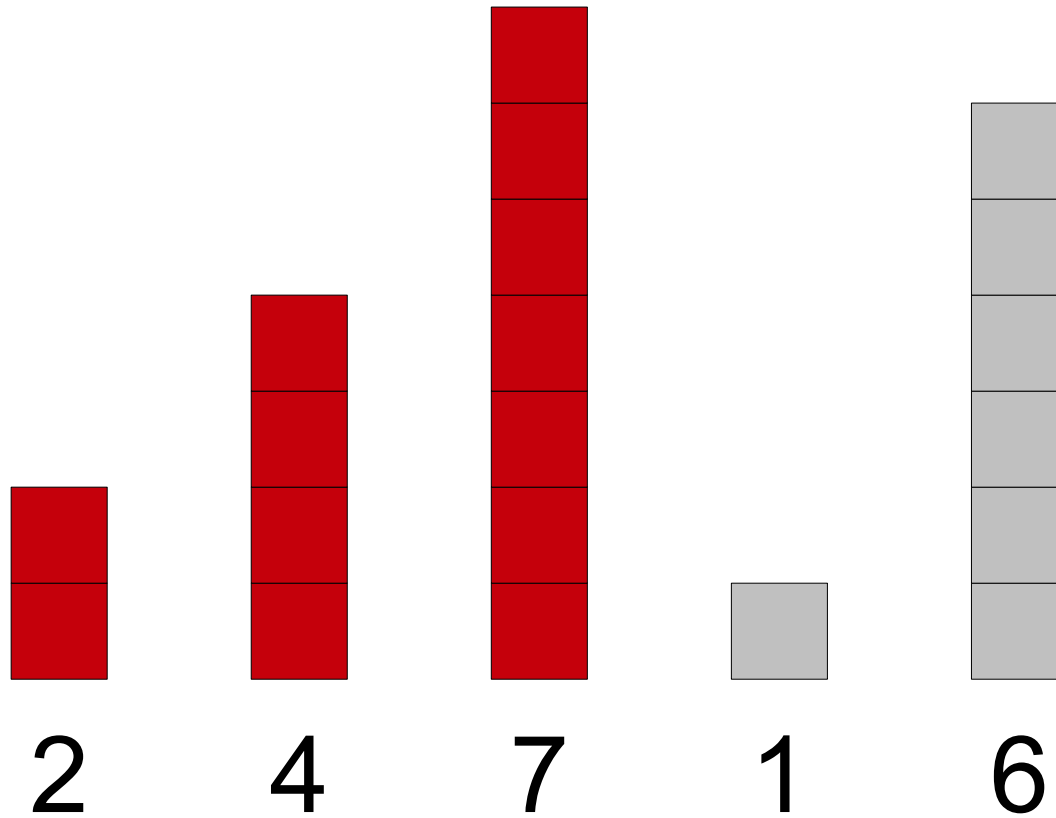
Another Idea: Insertion Sort



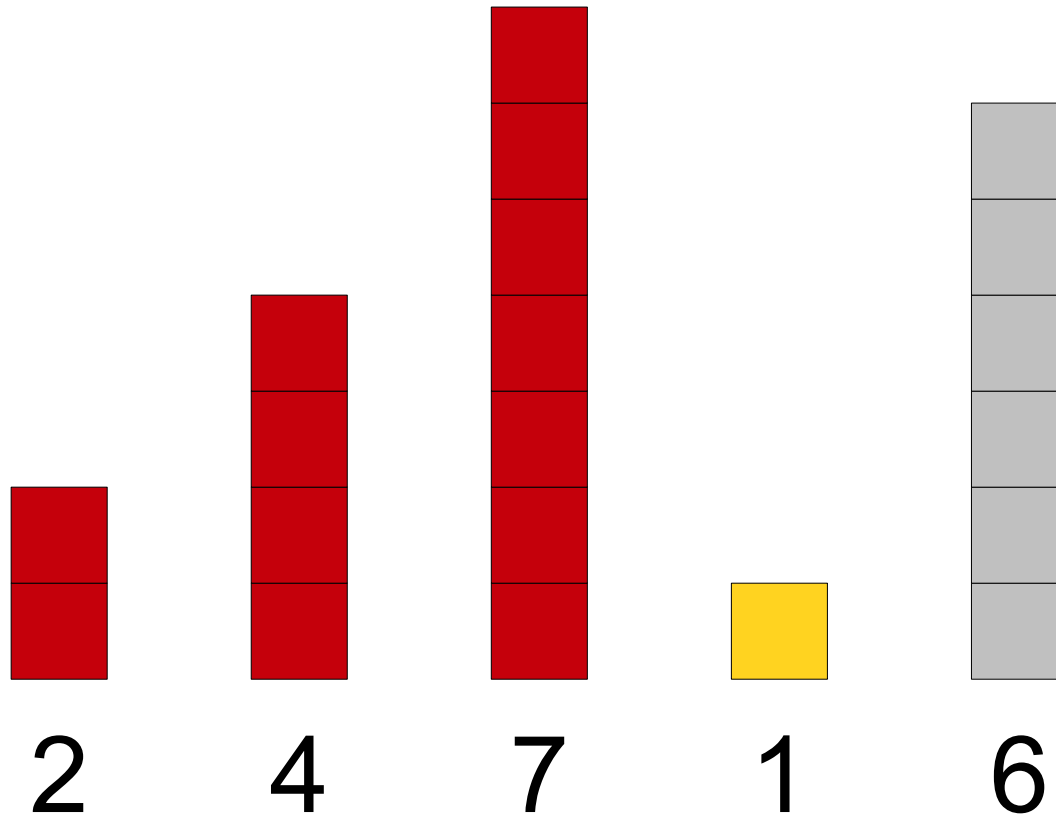
Another Idea: Insertion Sort



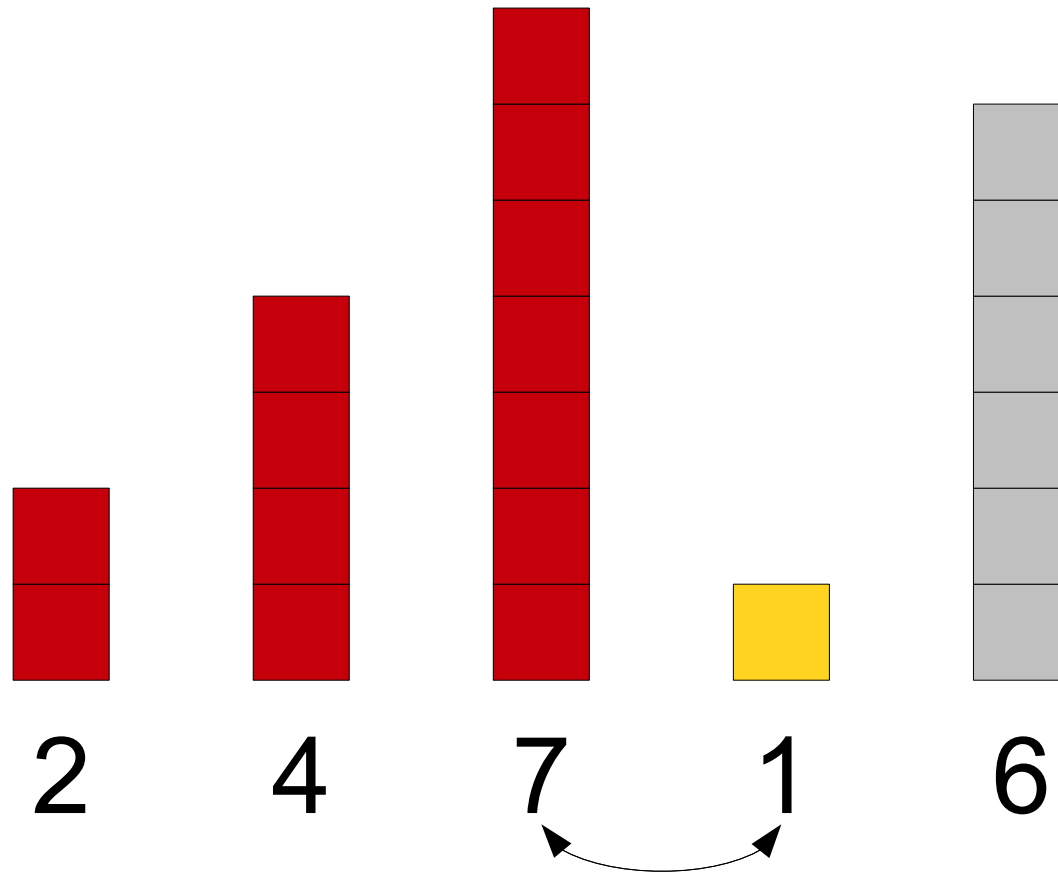
Another Idea: Insertion Sort



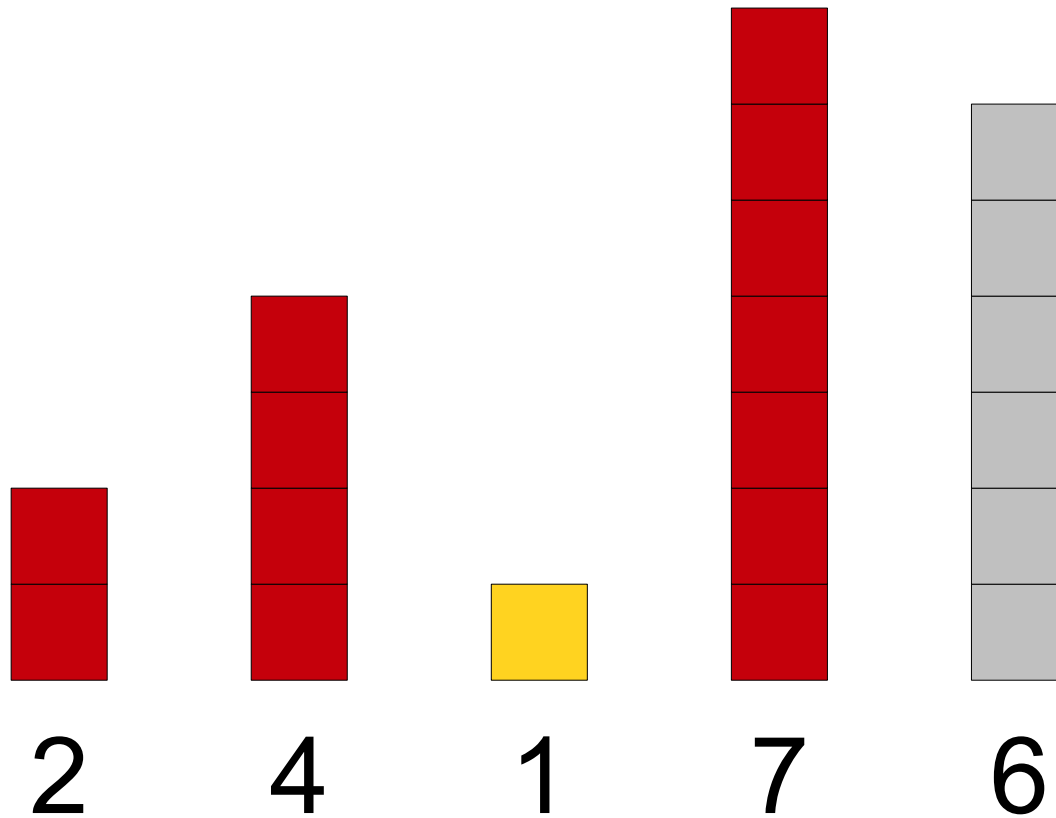
Another Idea: Insertion Sort



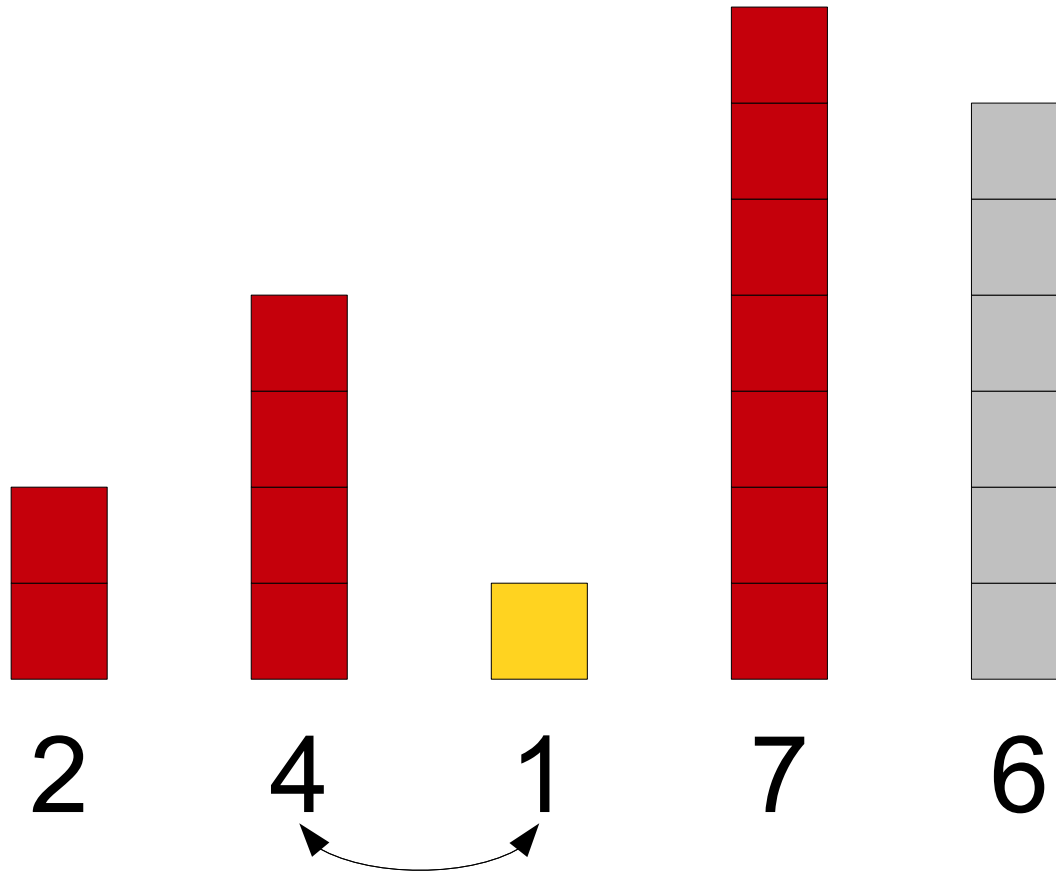
Another Idea: Insertion Sort



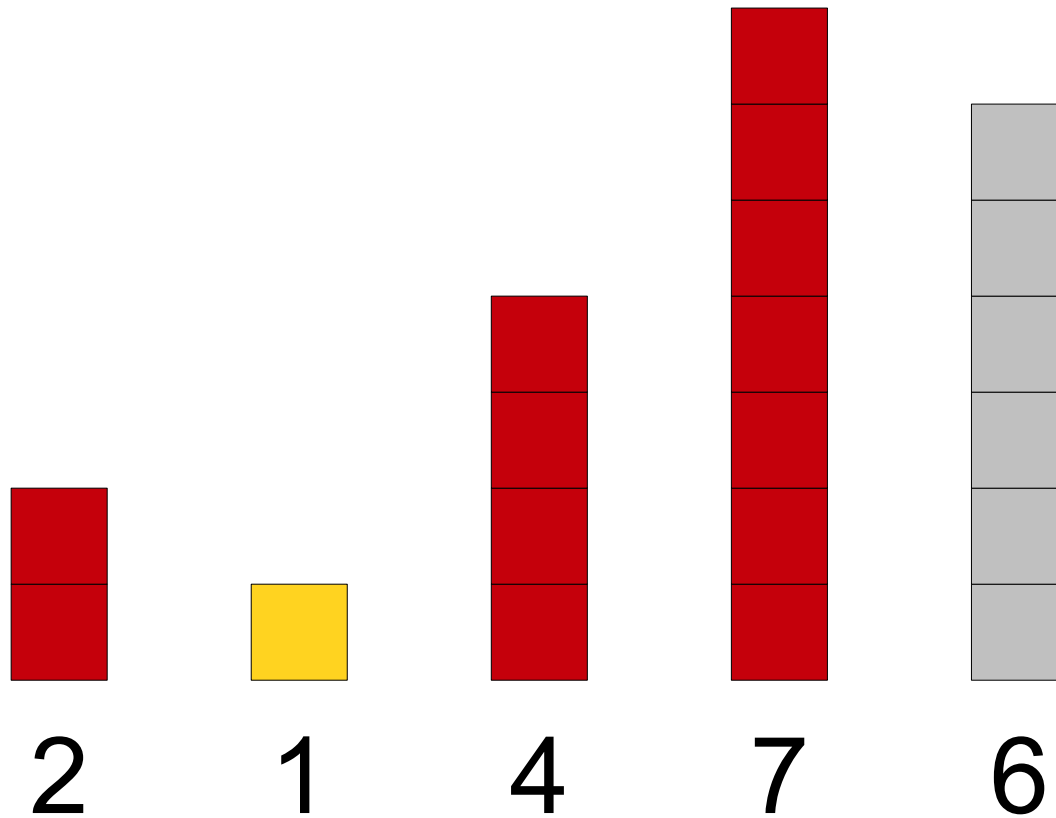
Another Idea: Insertion Sort



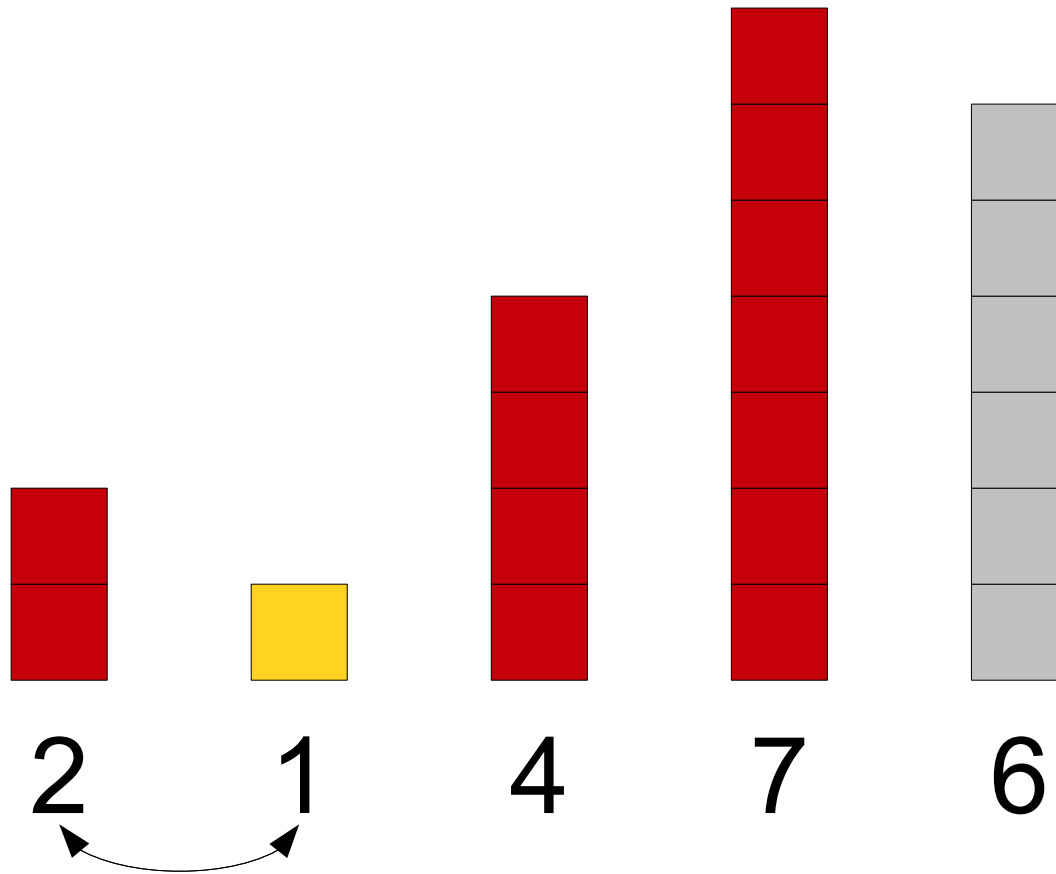
Another Idea: Insertion Sort



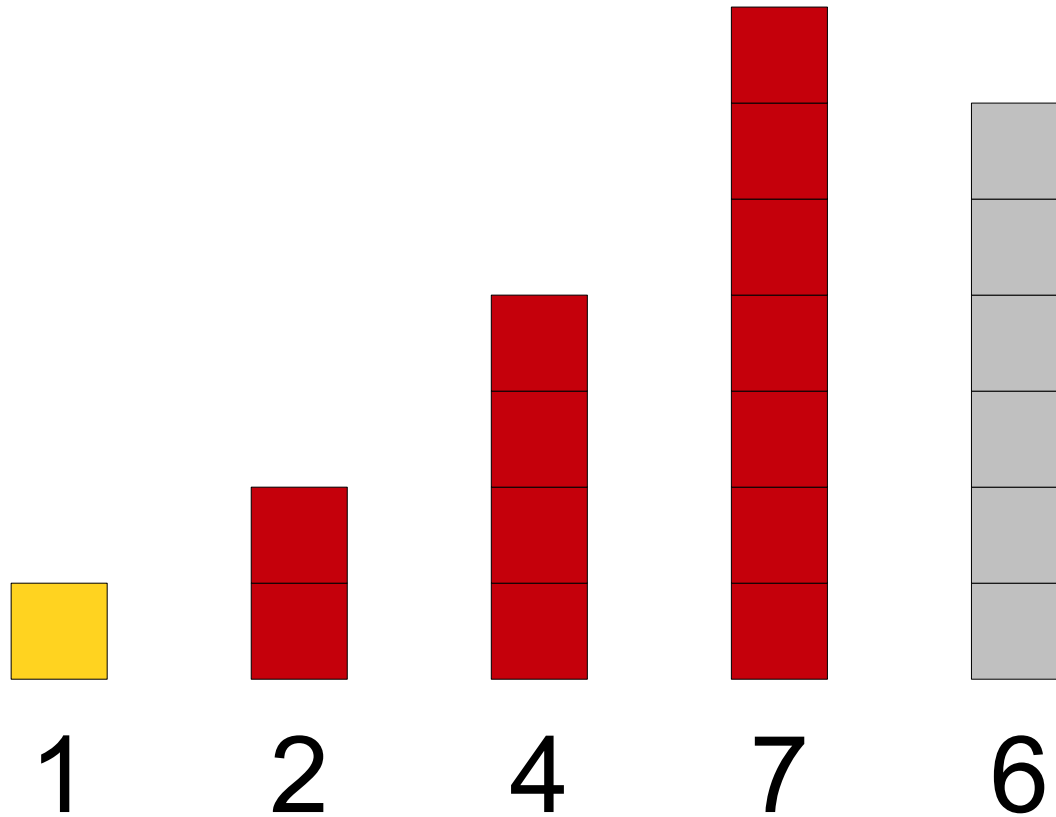
Another Idea: Insertion Sort



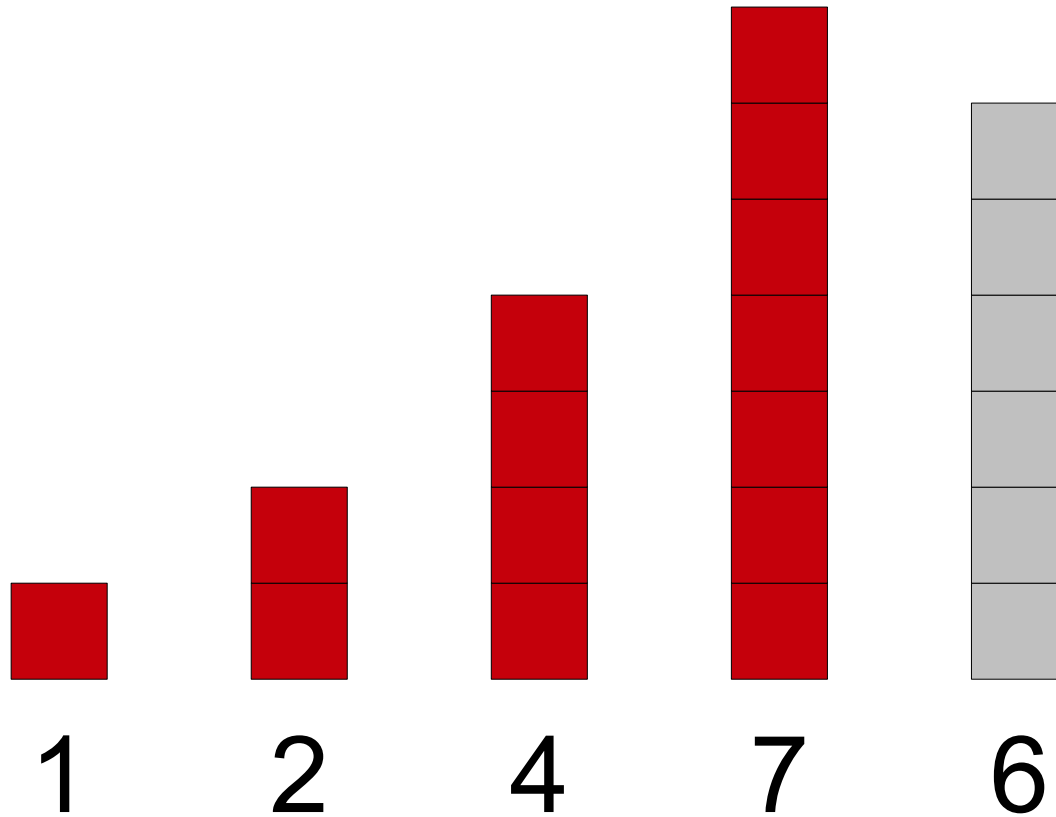
Another Idea: Insertion Sort



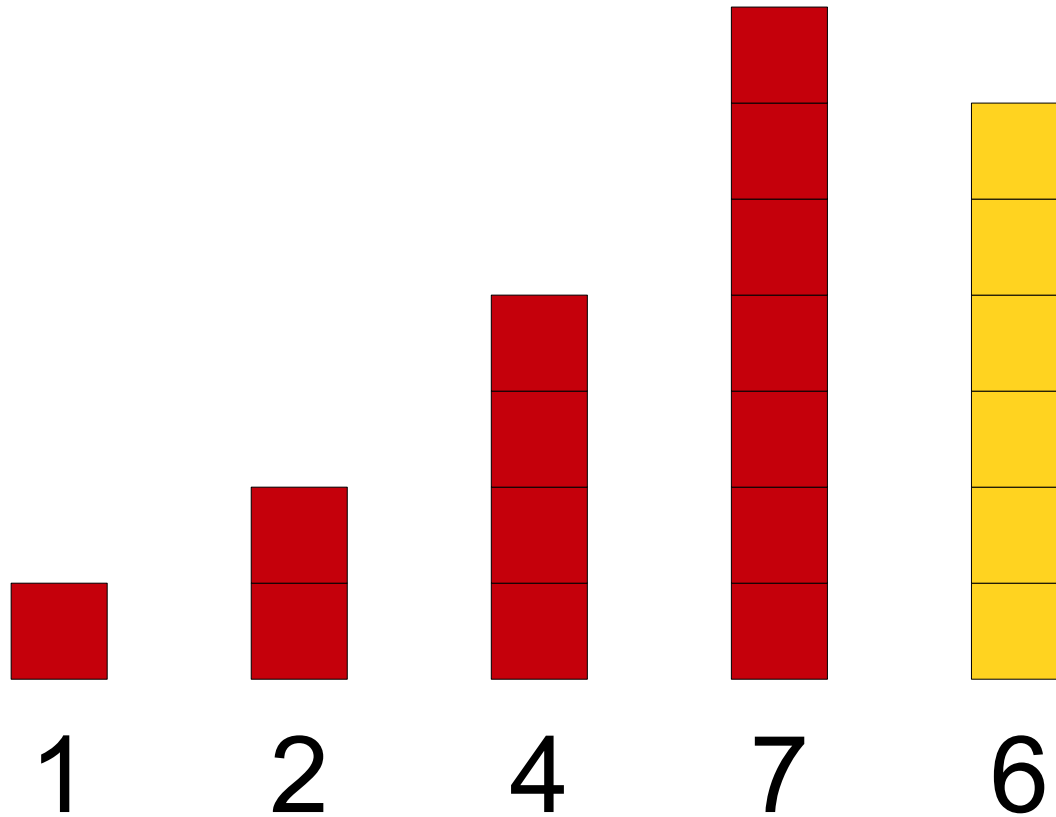
Another Idea: Insertion Sort



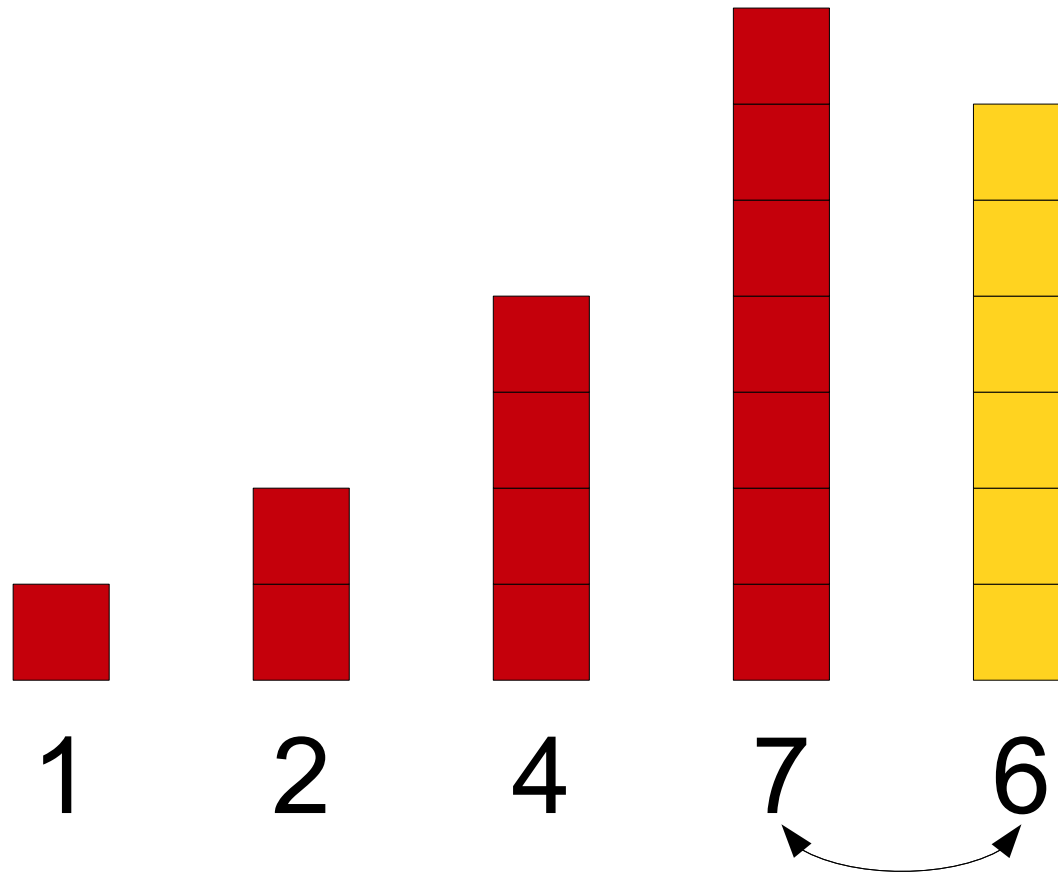
Another Idea: Insertion Sort



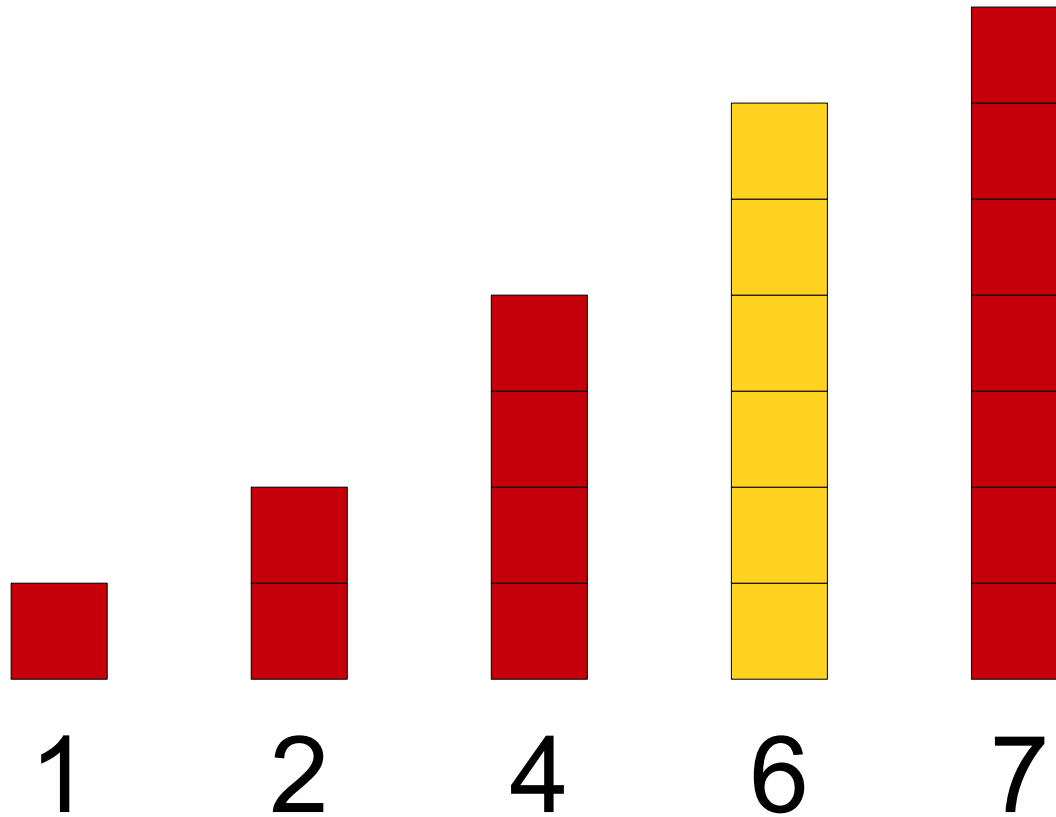
Another Idea: Insertion Sort



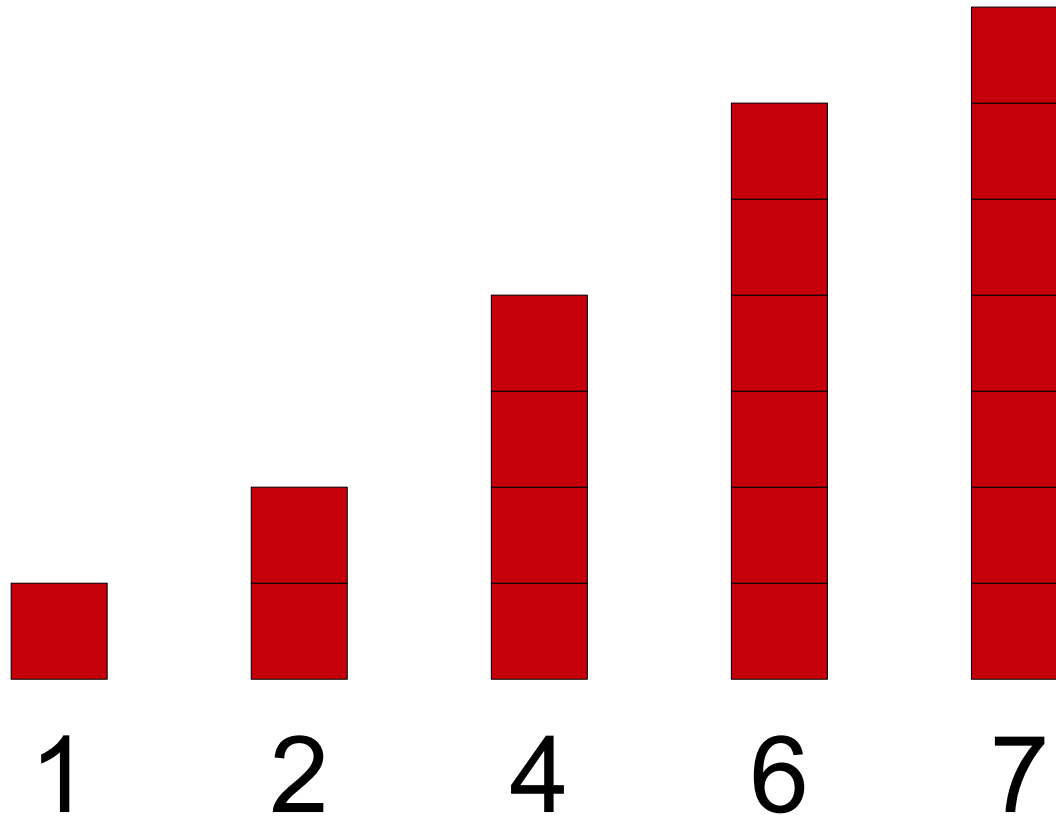
Another Idea: Insertion Sort



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]);
        }
    }
}
```

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]);
        }
    }
}
```

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]);
        }
    }
}
```

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]);
        }
    }
}
```

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]);
        }
    }
}
```

$O(n^2)$

Notes on Insertion Sort

- Insertion sort is $O(n^2)$ in the worst case.
- How about the best case?
- Only $O(n)$.
- Insertion sort is almost always faster than selection sort.

Selection Sort vs Insertion Sort

- Both algorithms are $O(n^2)$ in the worst case.
- Selection sort does more work at the beginning.
- Insertion sort does more work at the end.

A Note on $O(n^2)$

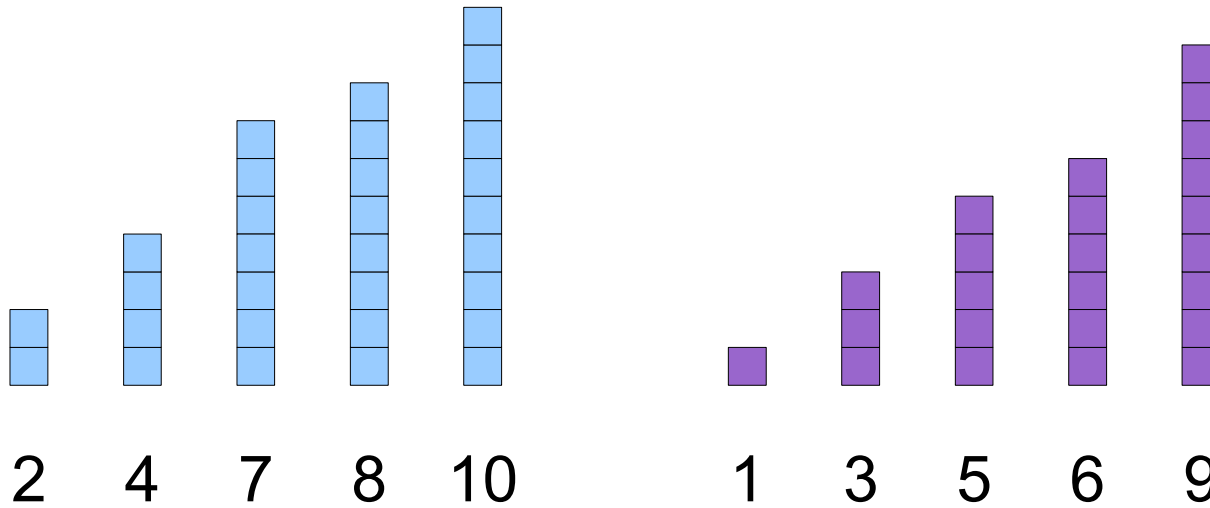
- If an algorithm is $O(n^2)$, what happens to the runtime if we double the input size?
 - Should go up by a factor of four: $(2n)^2 = 4n^2$
- If an algorithm is $O(n^2)$, what happens to the runtime if we *halve* the input size?
 - Should go *down* by a factor of four: $(\frac{1}{2}n)^2 = \frac{1}{4}n^2$

Breaking an $O(n^2)$ algorithm into **smaller pieces** and **solving those pieces independently** takes **less time** than the original algorithm!

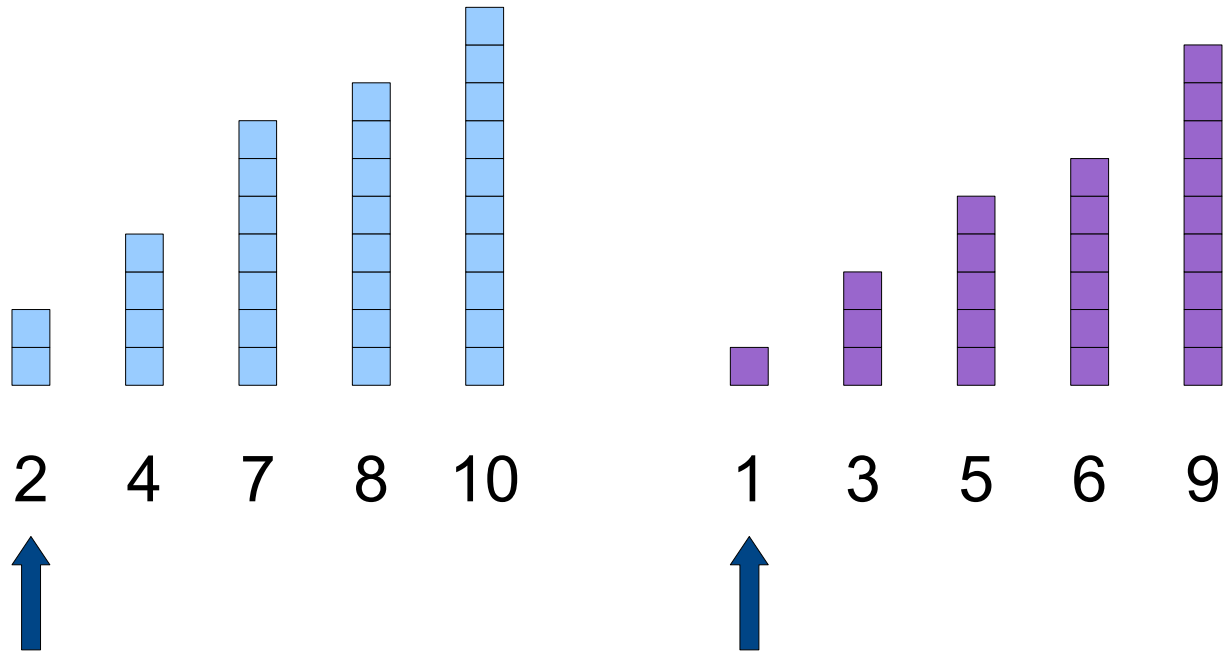
Can we use this to build a better sort?

The Key Insight: **Merge**

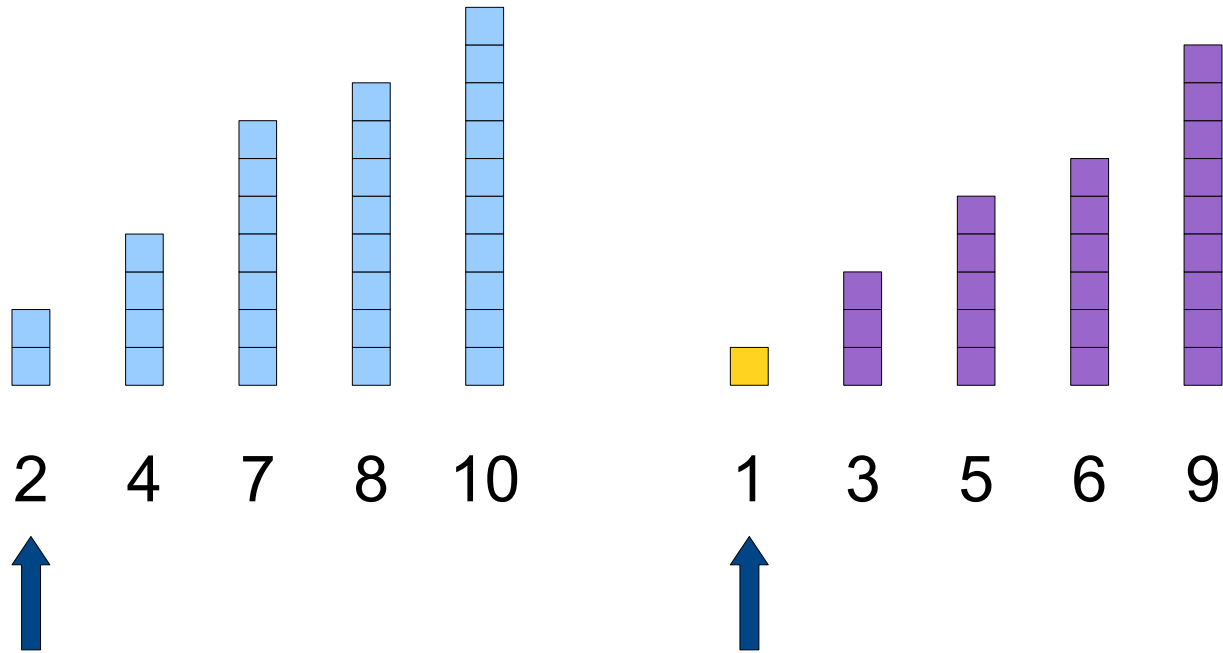
The Key Insight: Merge



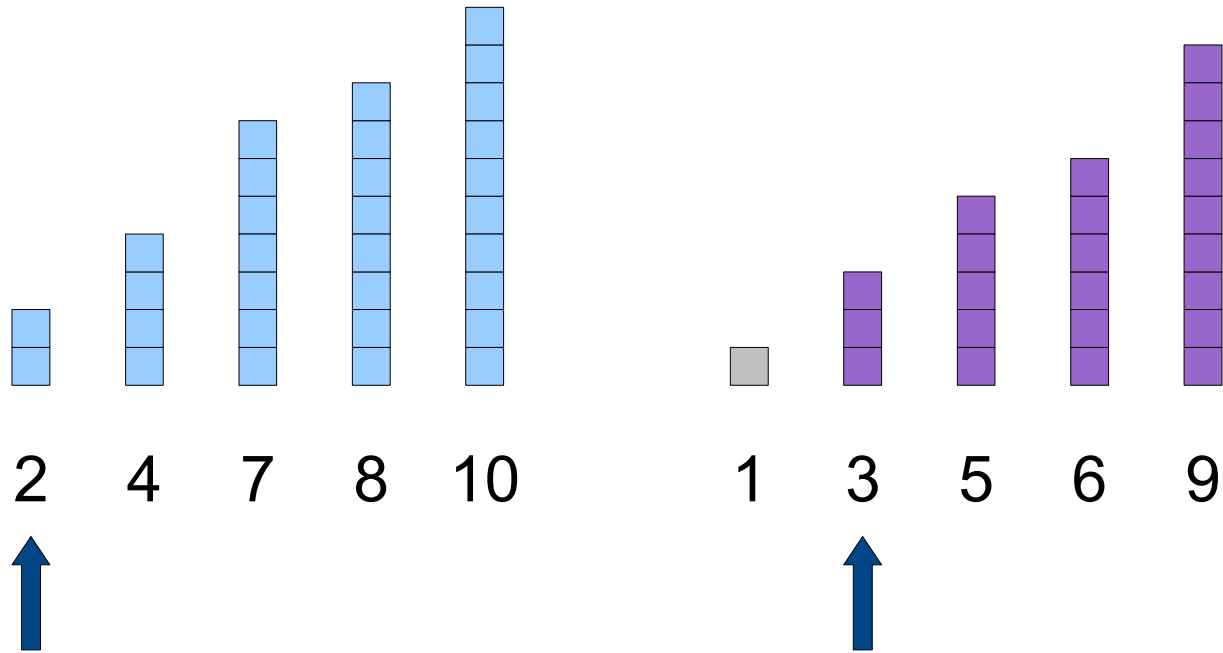
The Key Insight: Merge



The Key Insight: Merge

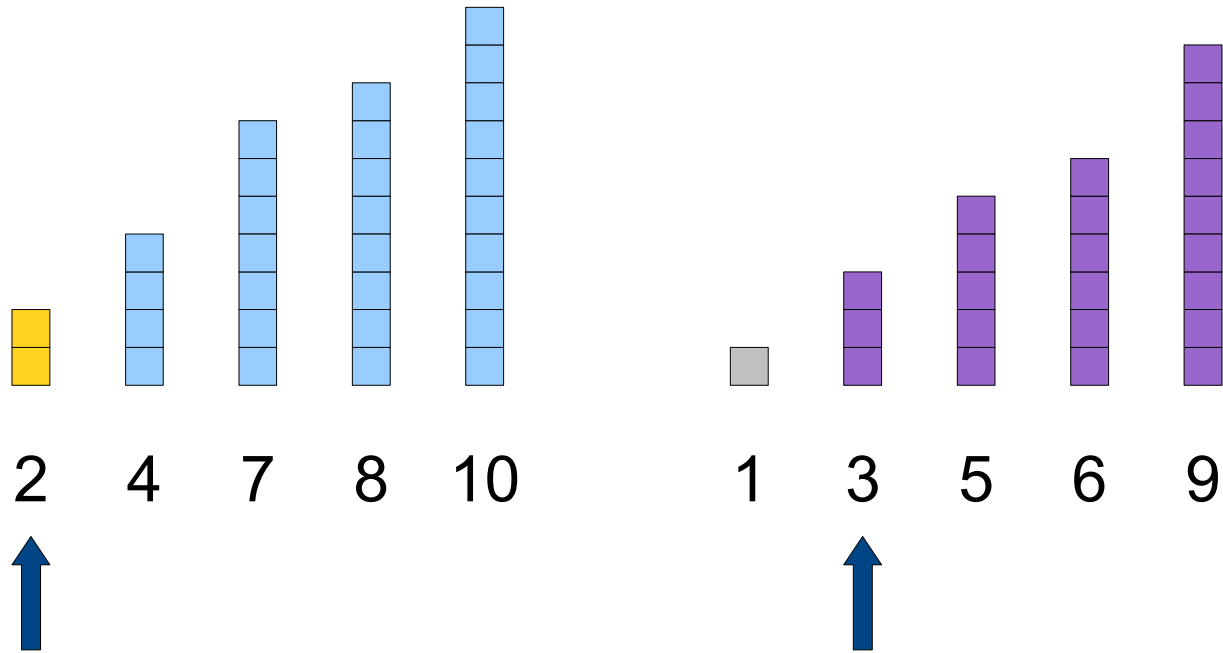


The Key Insight: Merge



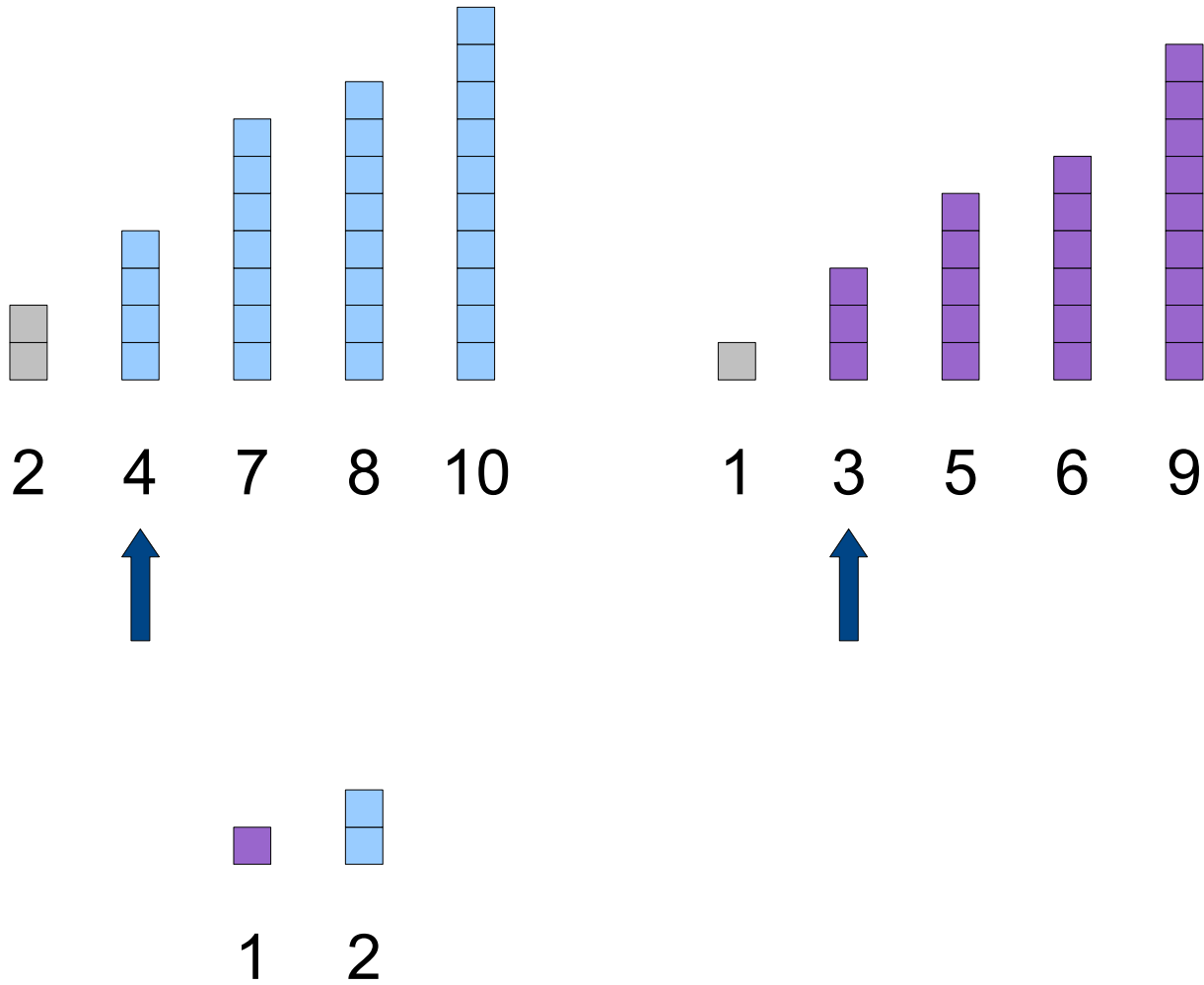
1

The Key Insight: Merge

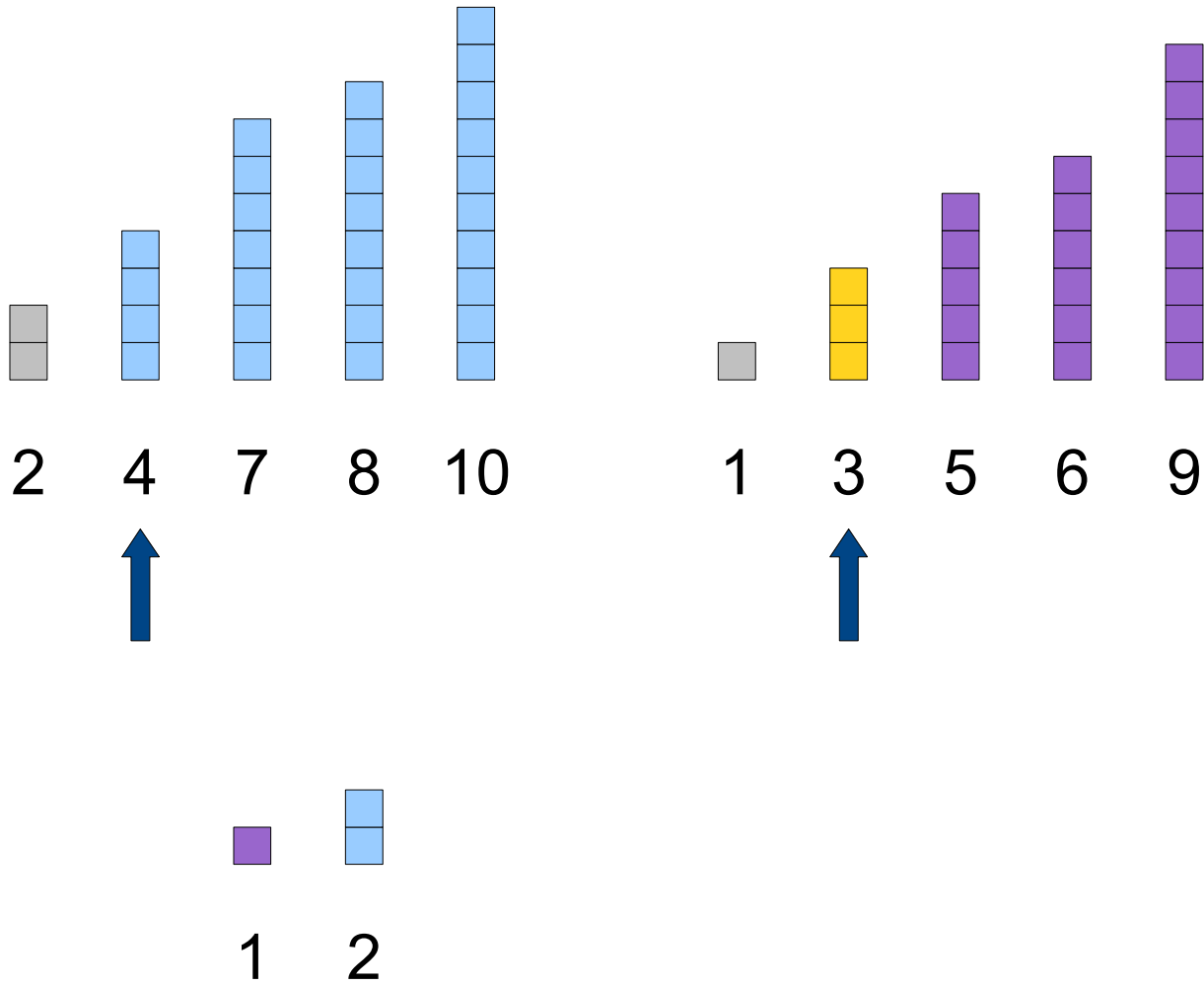


1

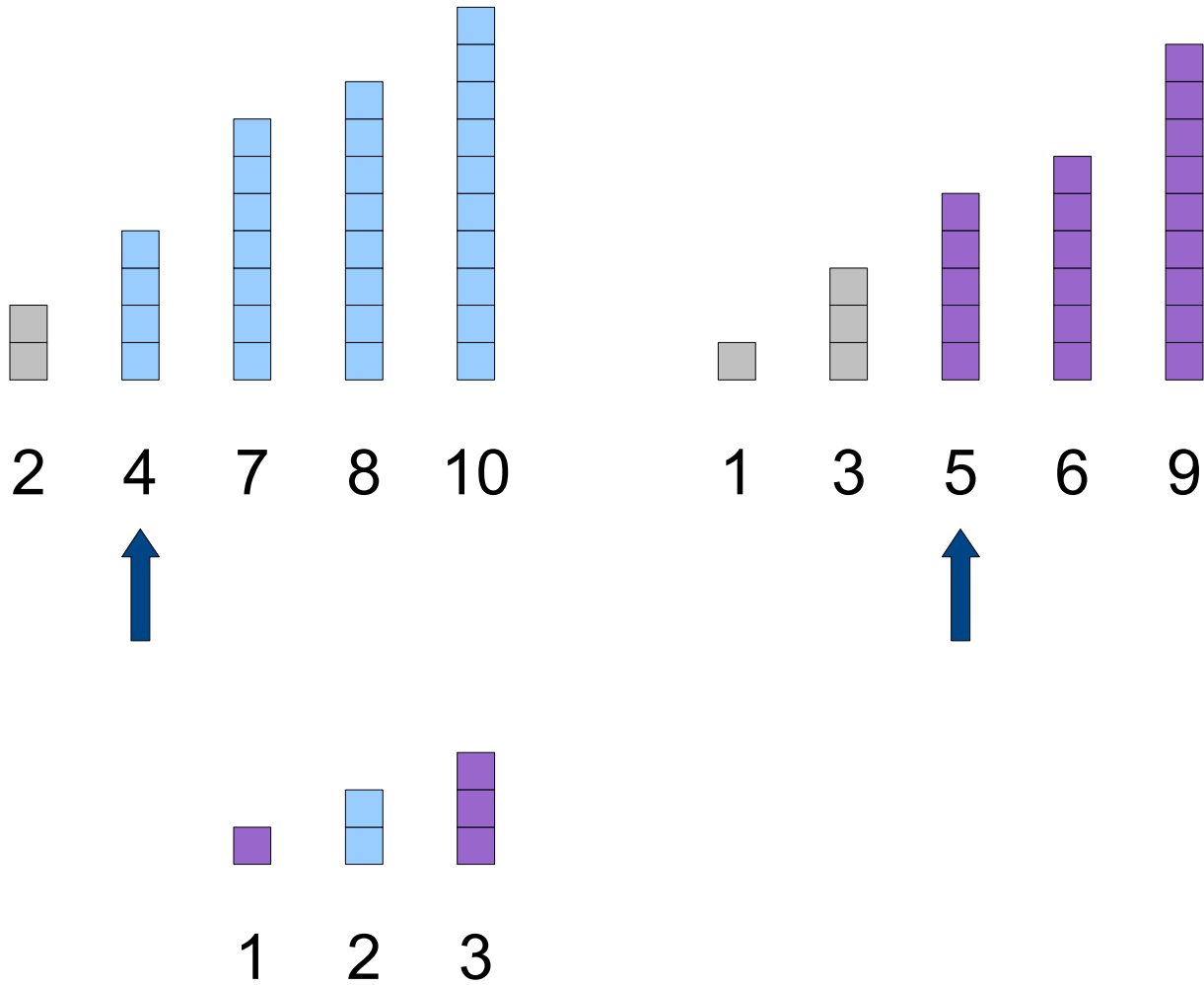
The Key Insight: Merge



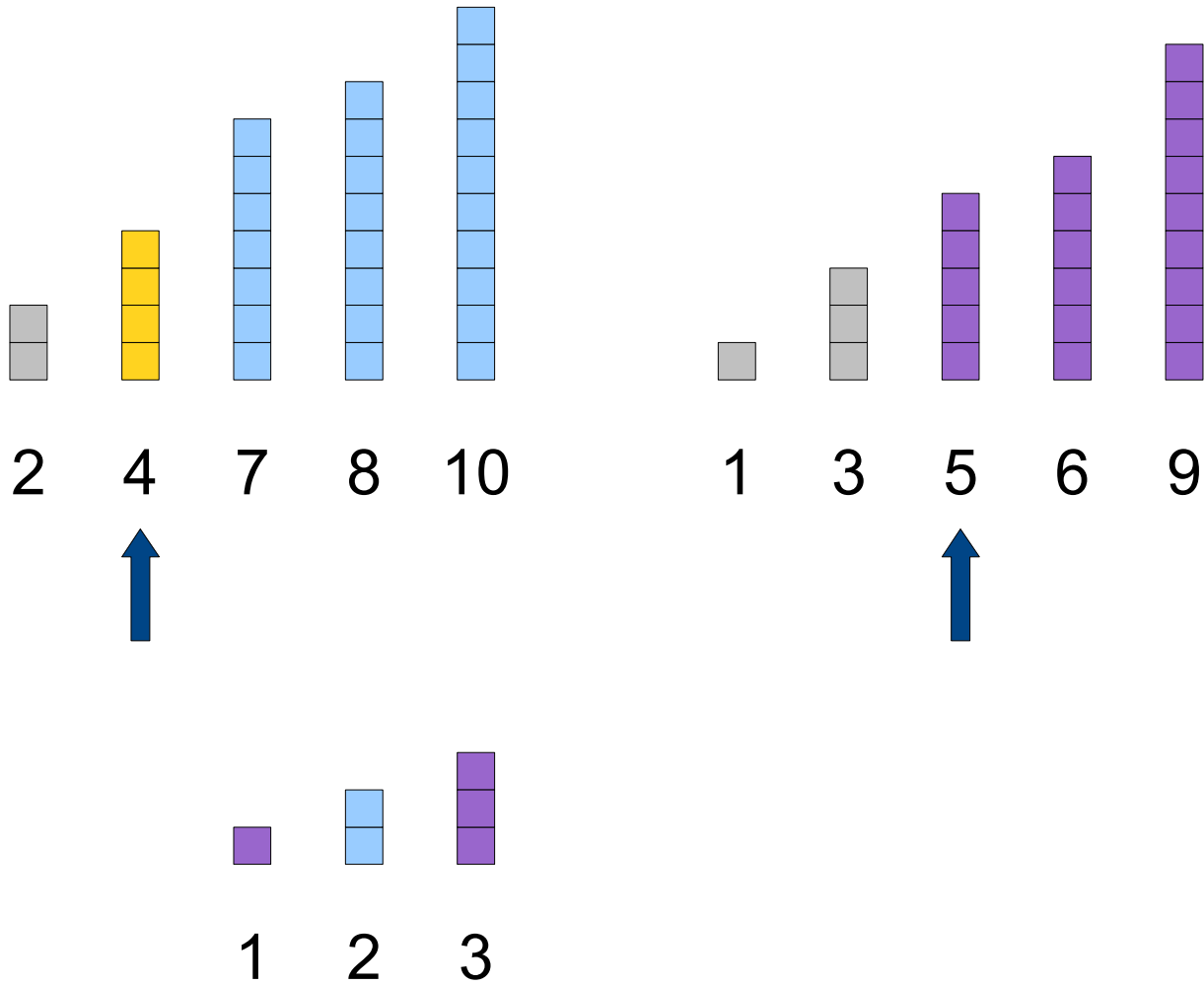
The Key Insight: Merge



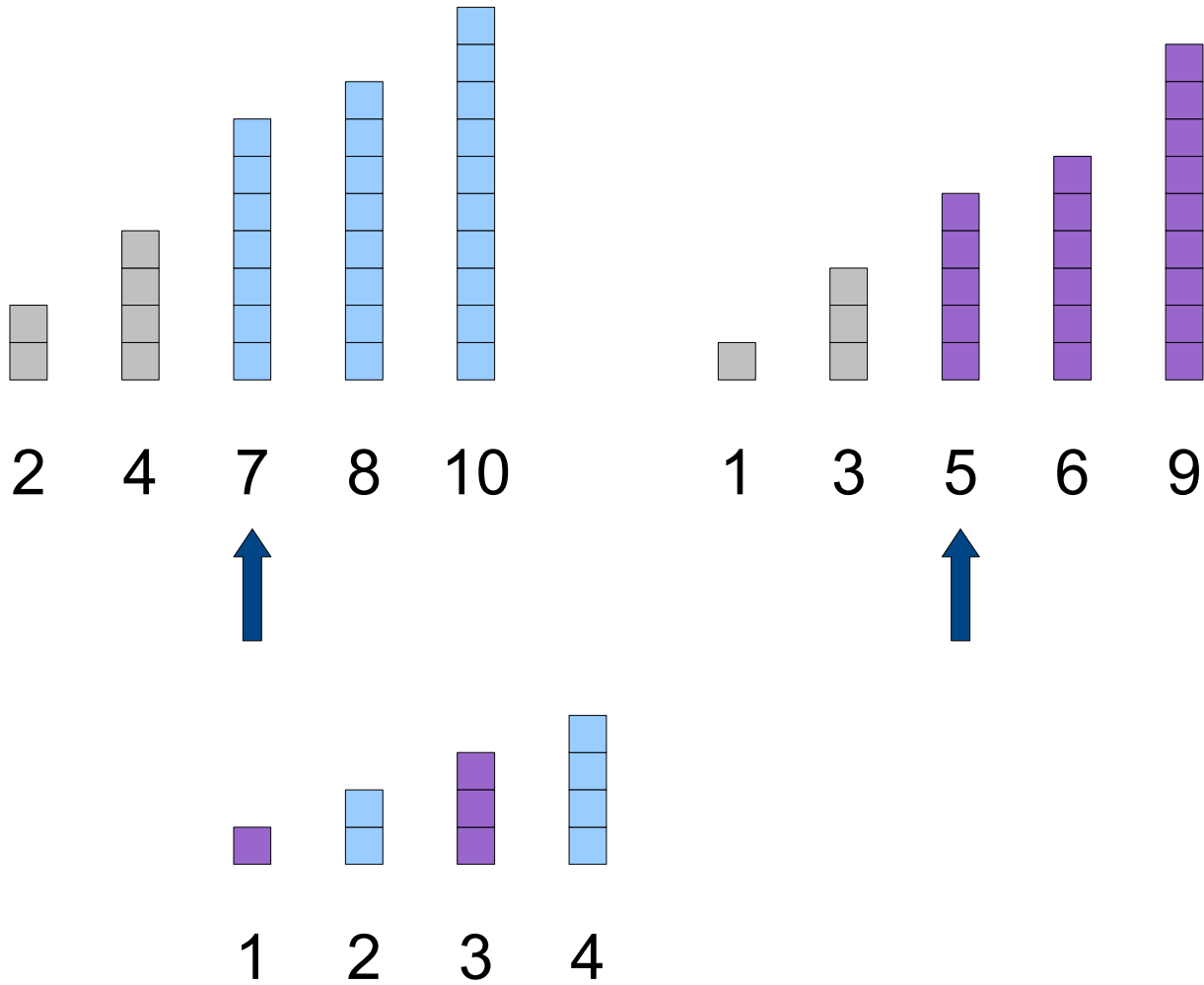
The Key Insight: Merge



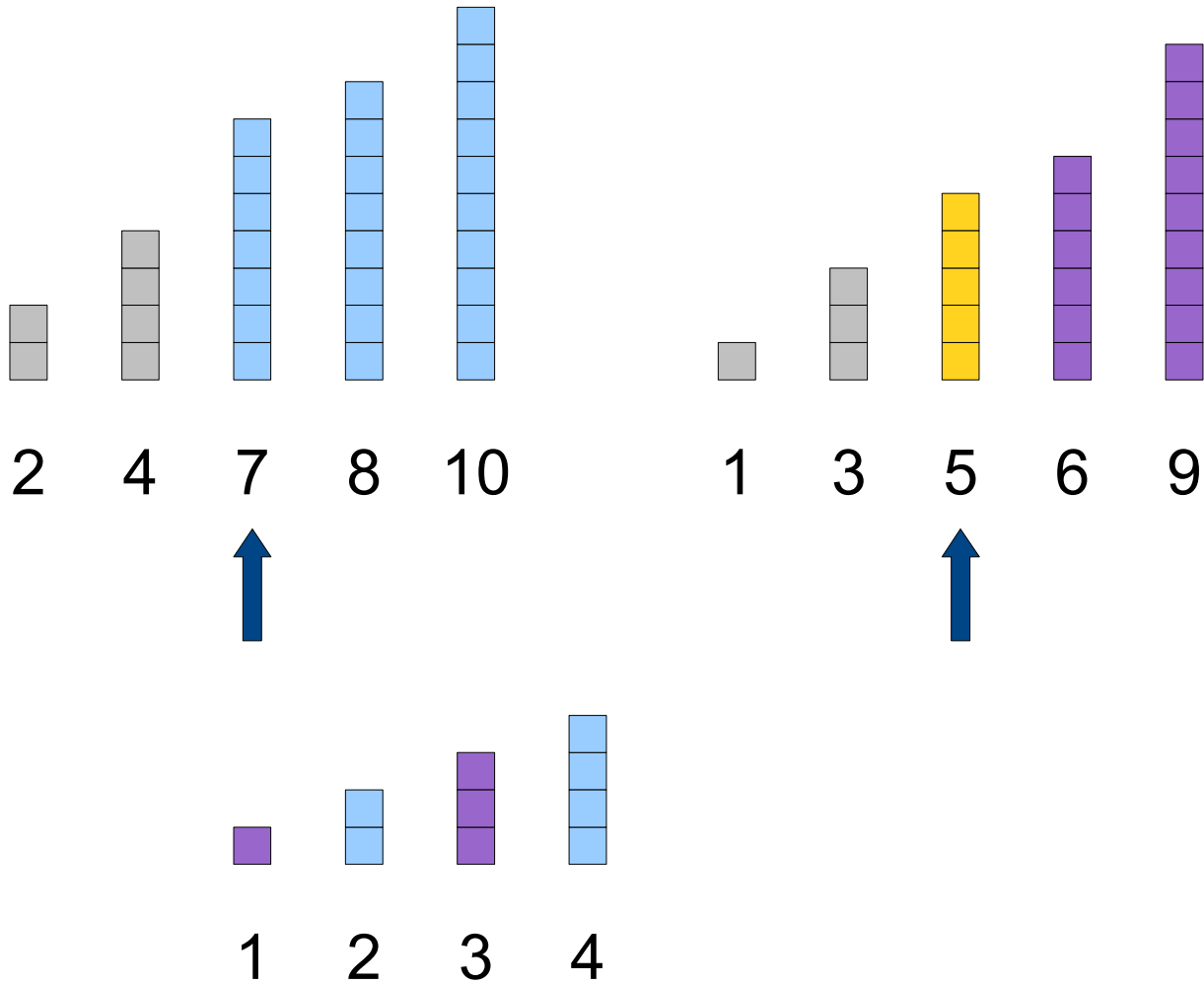
The Key Insight: Merge



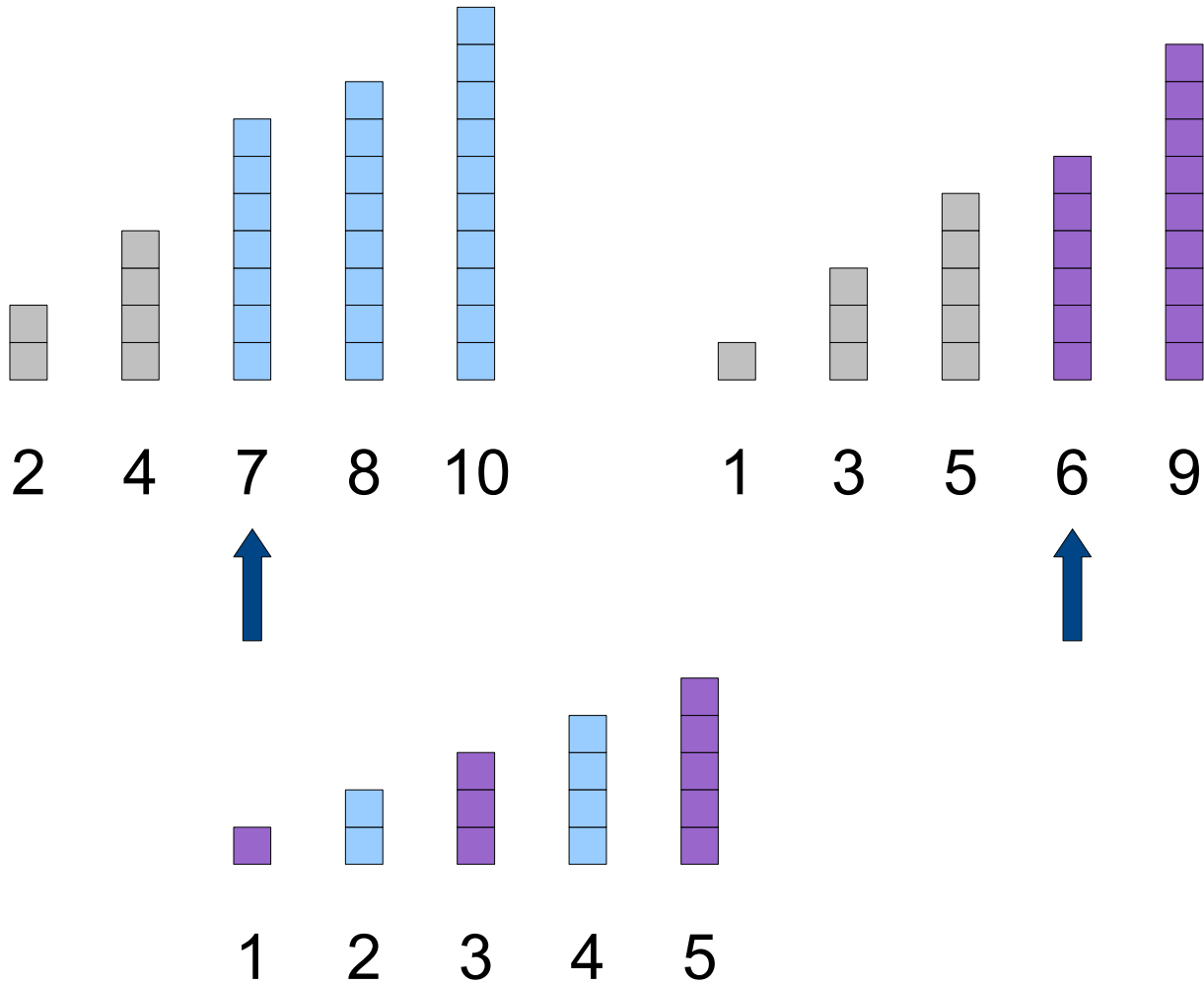
The Key Insight: Merge



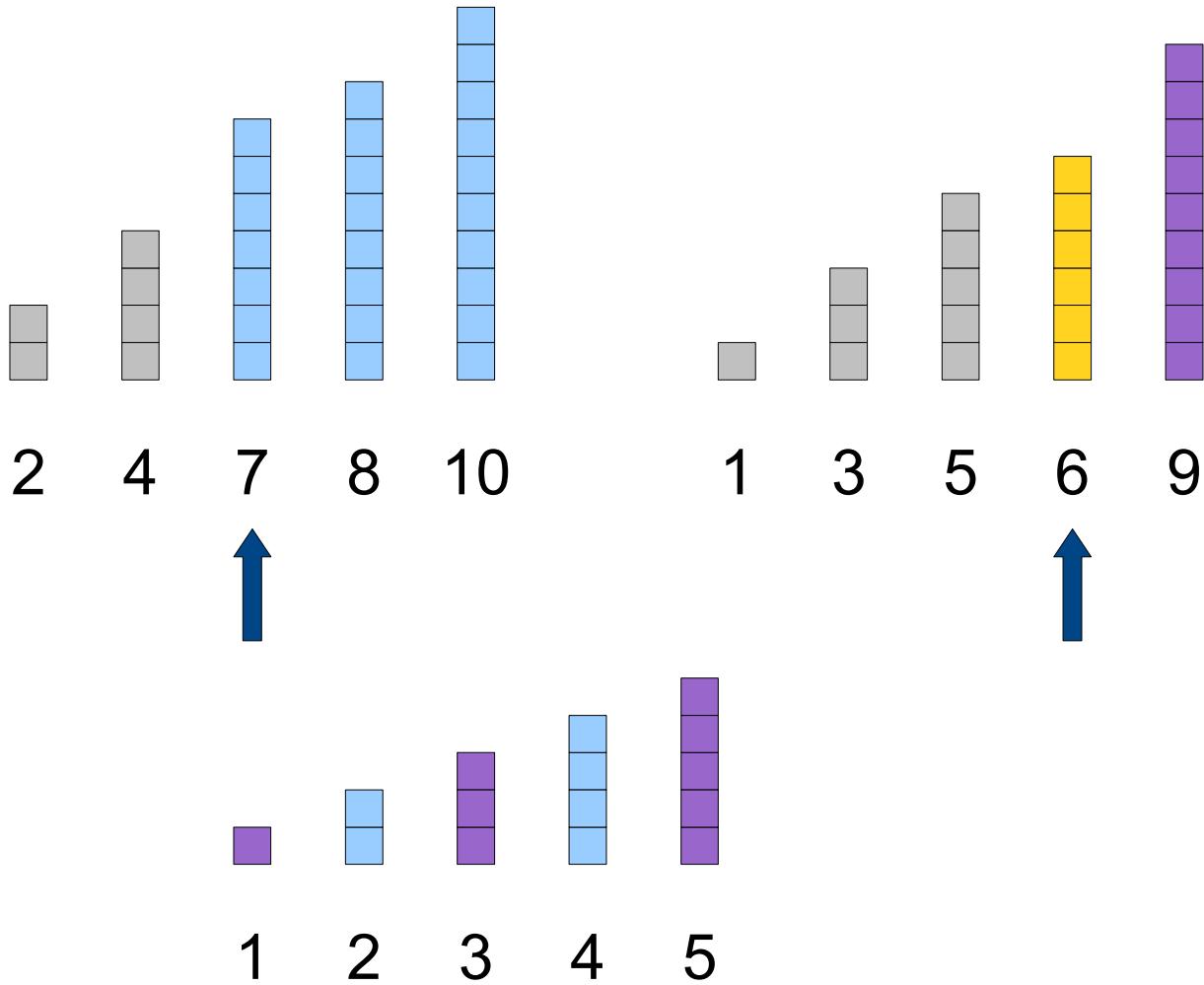
The Key Insight: Merge



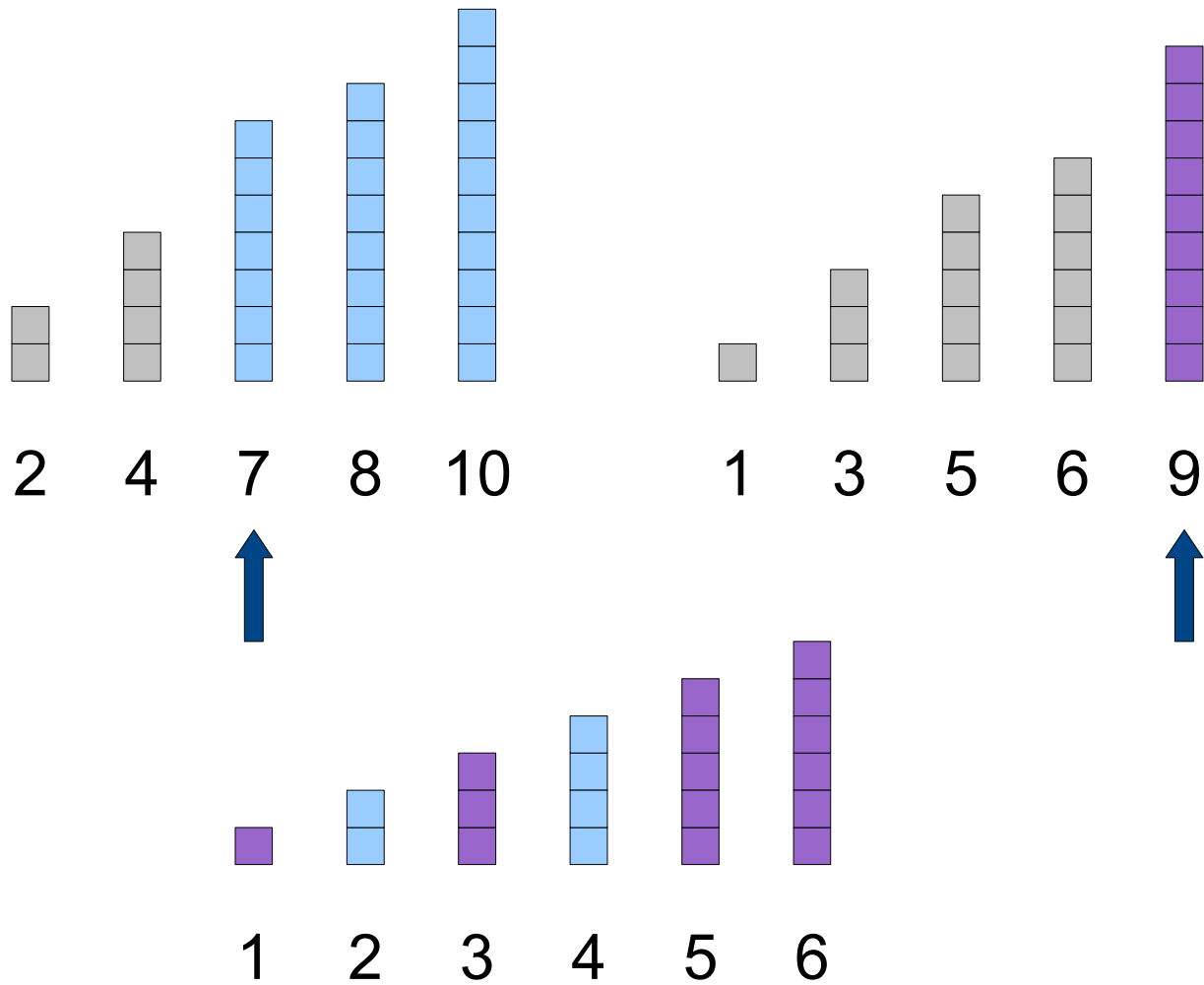
The Key Insight: Merge



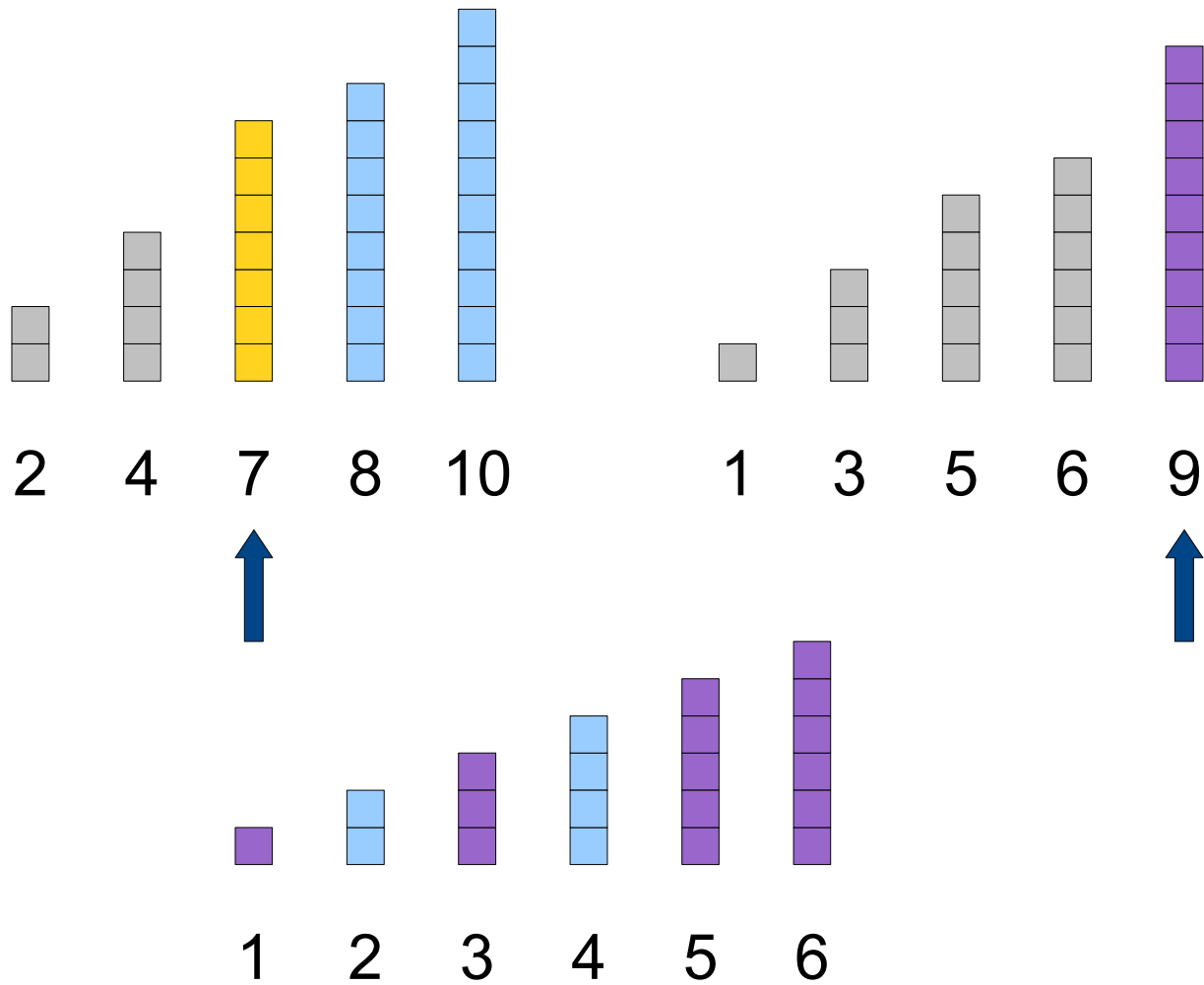
The Key Insight: Merge



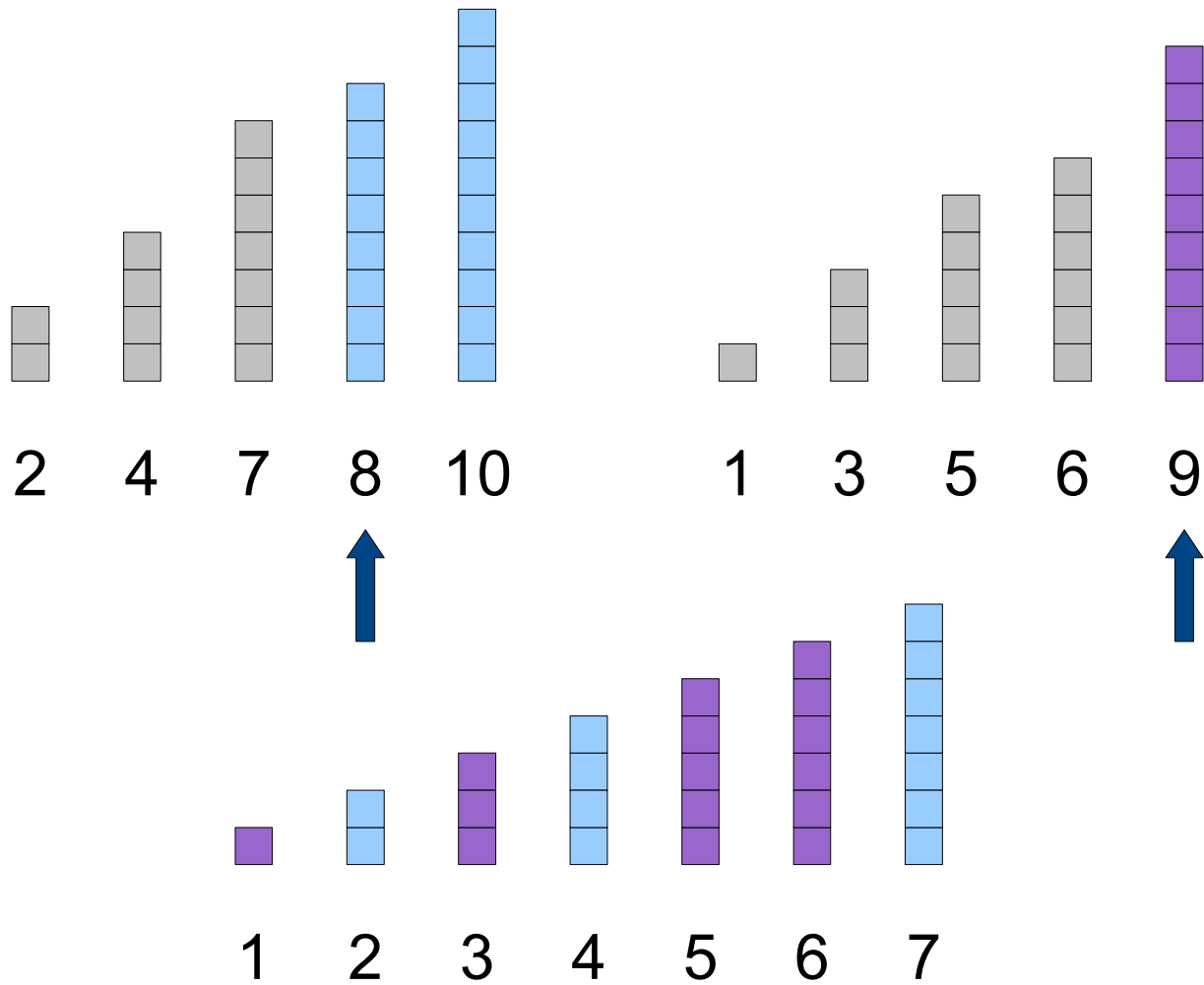
The Key Insight: Merge



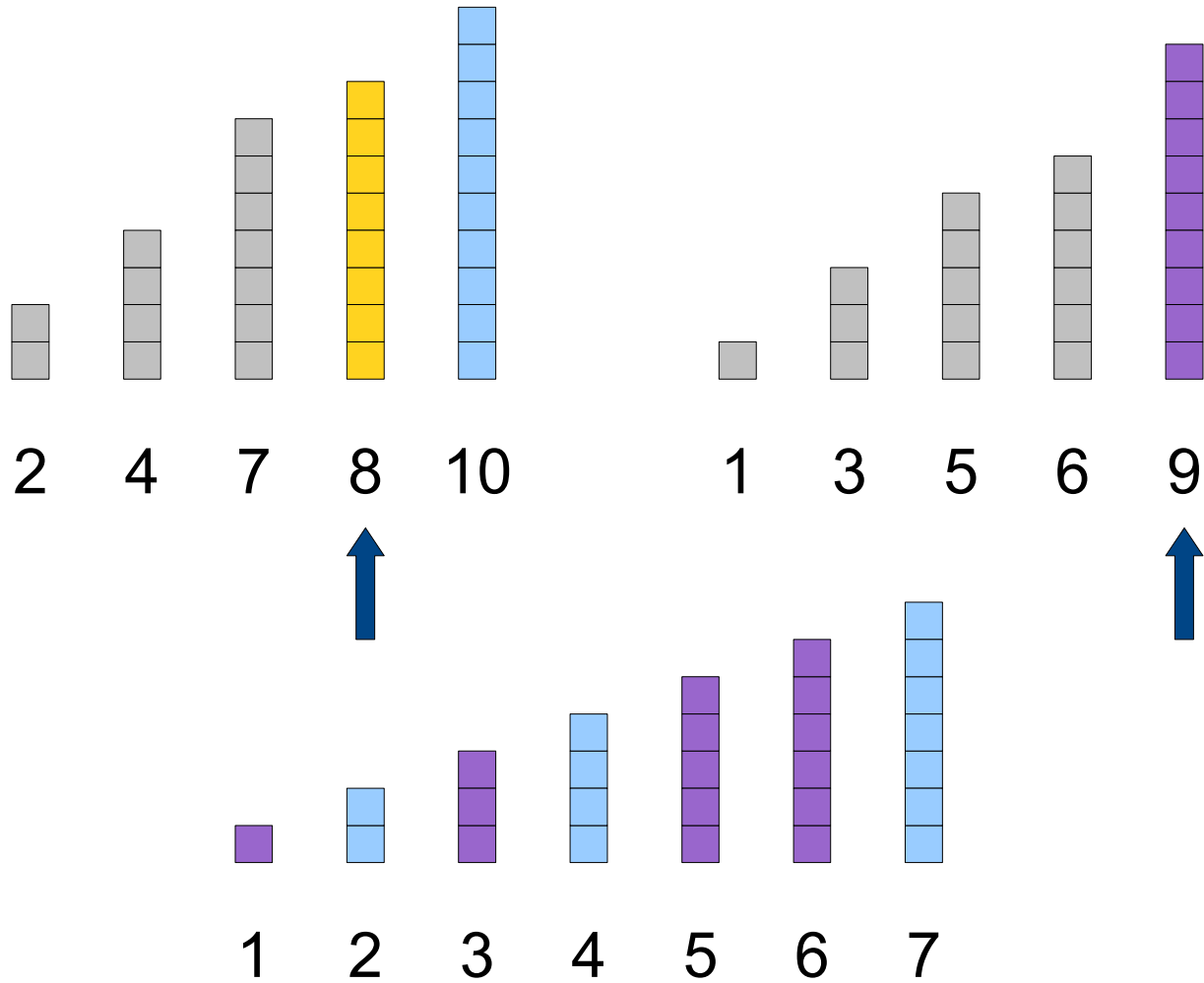
The Key Insight: Merge



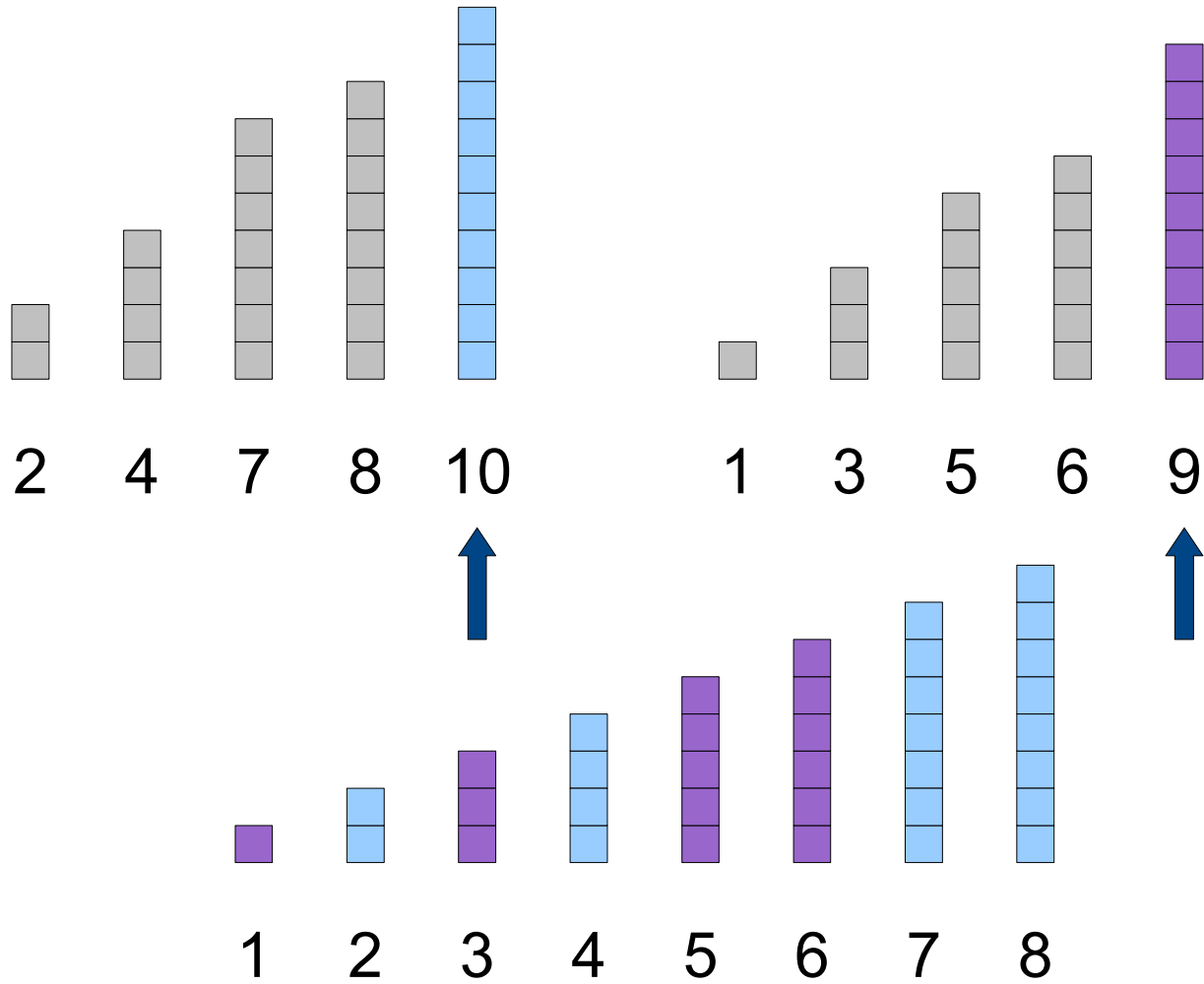
The Key Insight: Merge



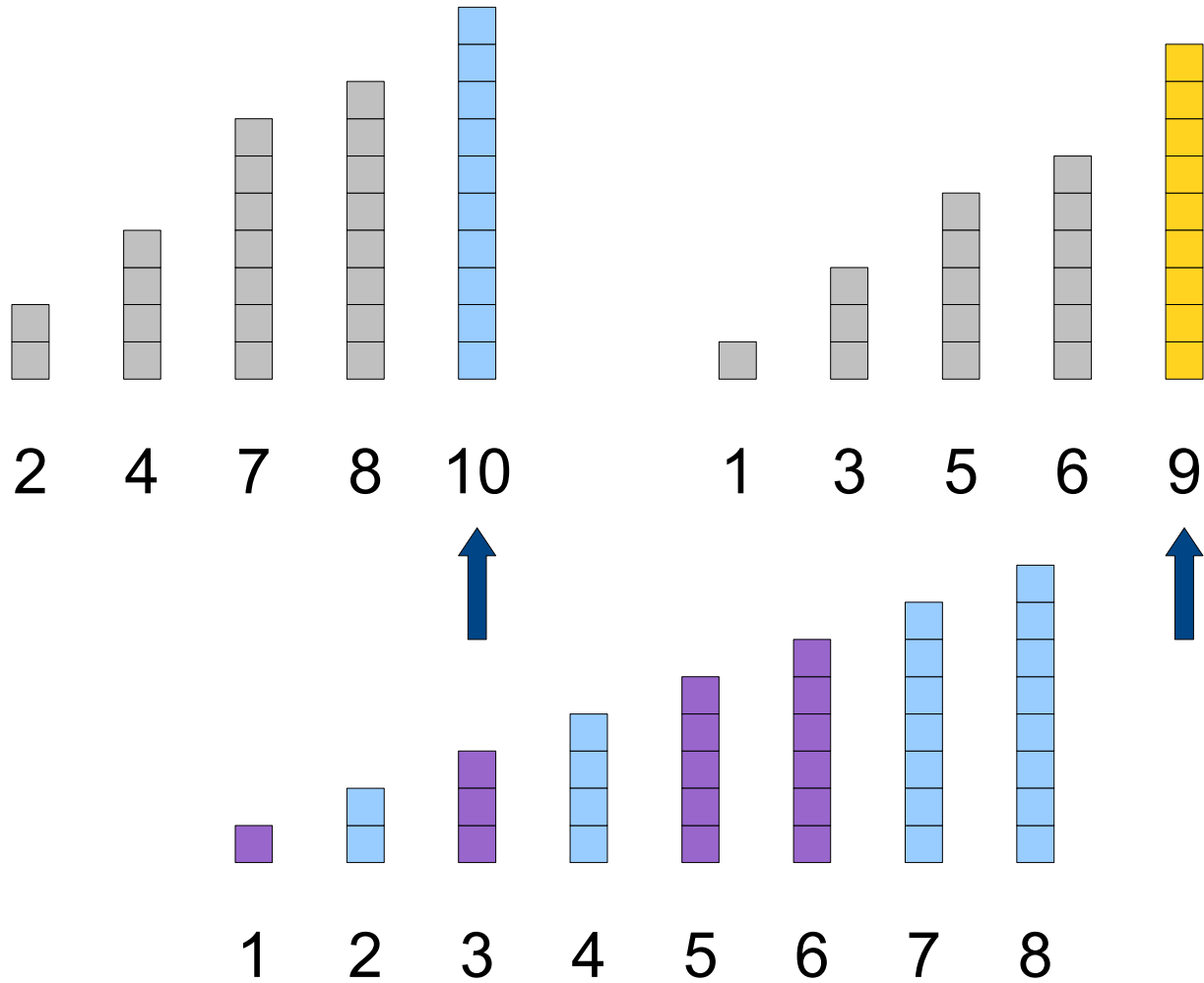
The Key Insight: Merge



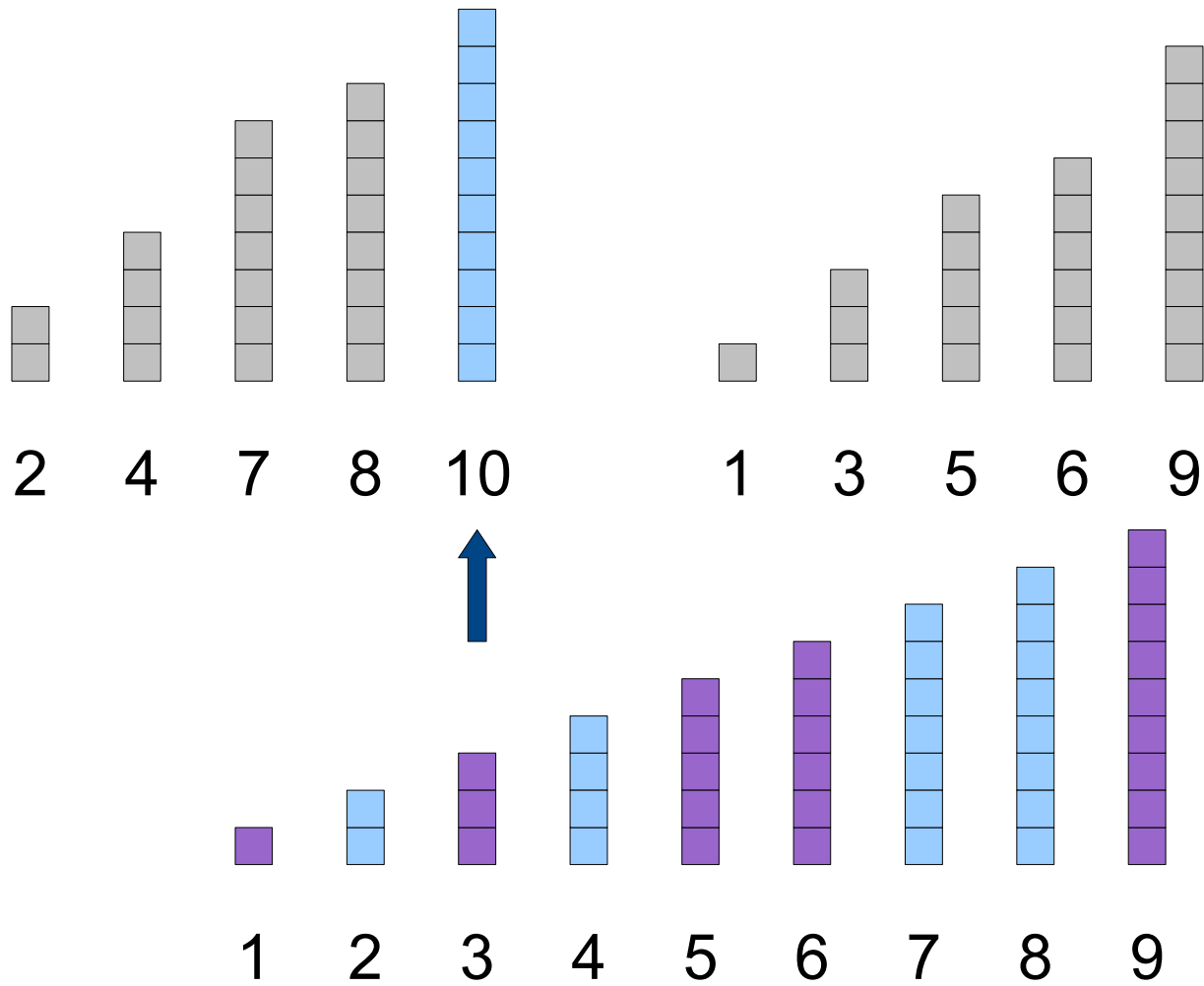
The Key Insight: Merge



The Key Insight: Merge



The Key Insight: Merge



The Key Insight: Merge

