

Algorithmic Analysis and Sorting, Part Two

CS106B
Winter 2009-2010

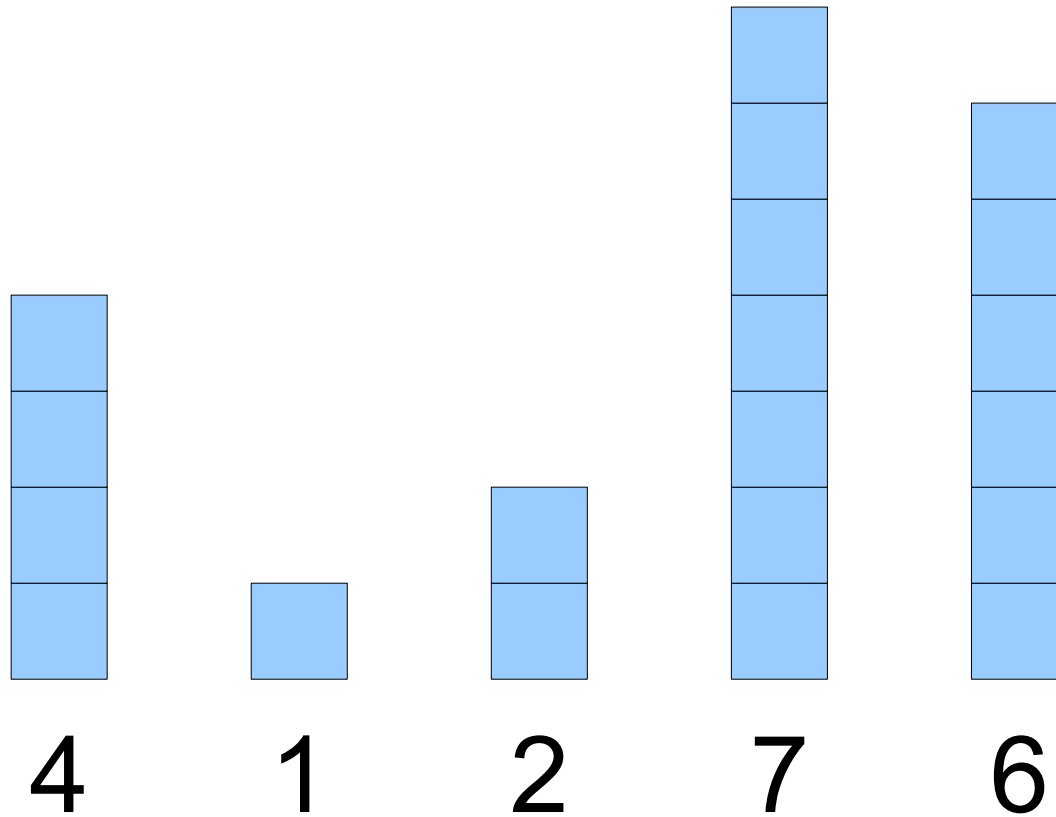
Previously on CS106B

Big-O Notation

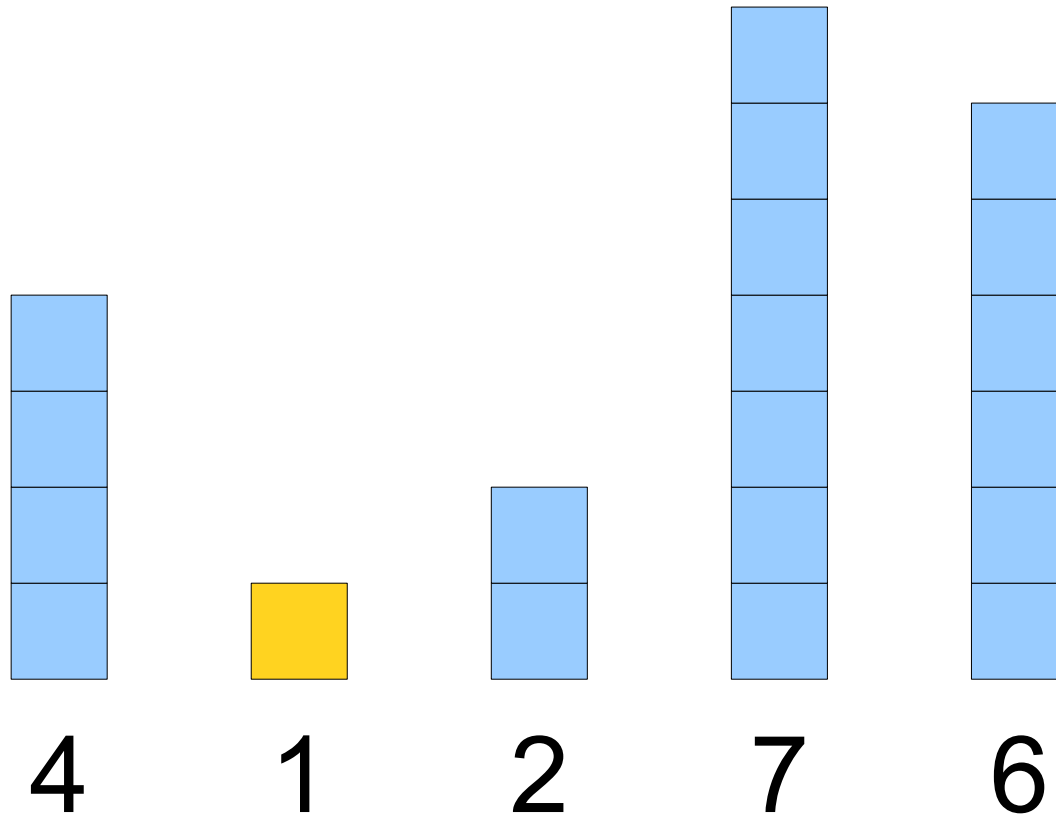
- Characterizes the long-term growth of a function.
- Drop all but the dominant term, ignore constants.
- Examples:
 - $6n + 22 = O(n)$
 - $n^2 + 137n = O(n^2)$

Selection Sort

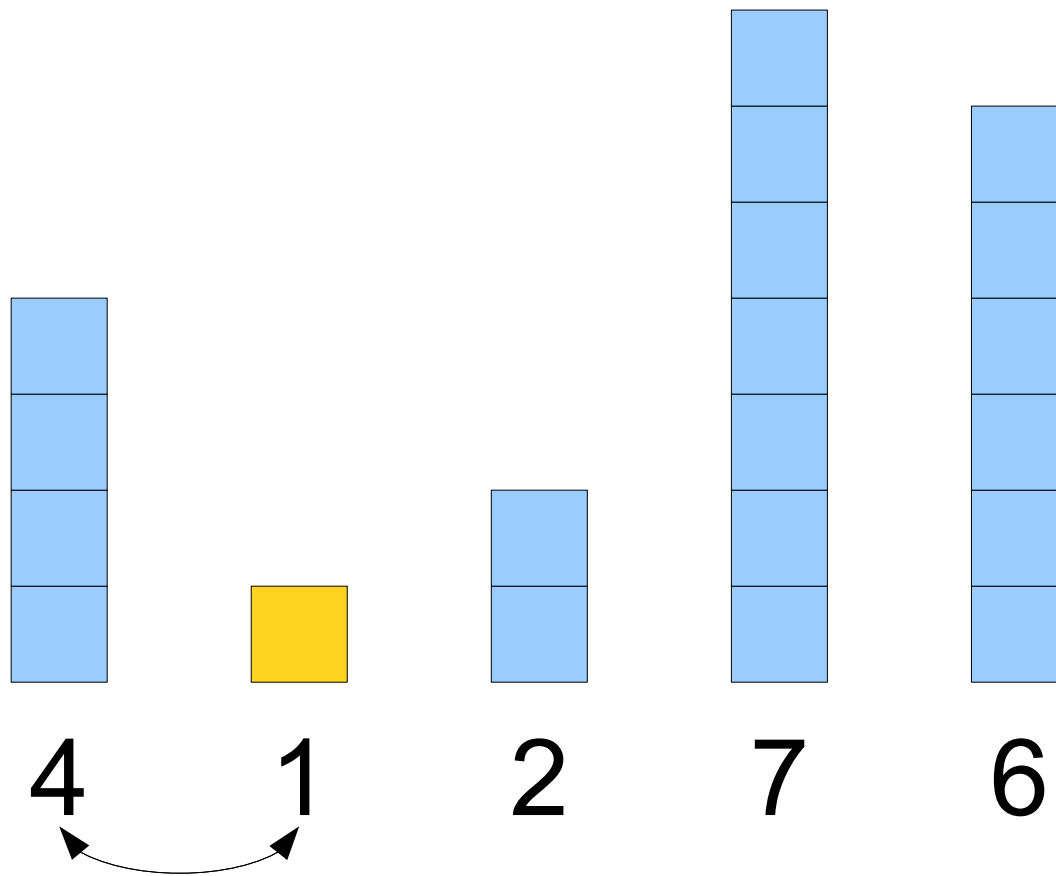
Selection Sort



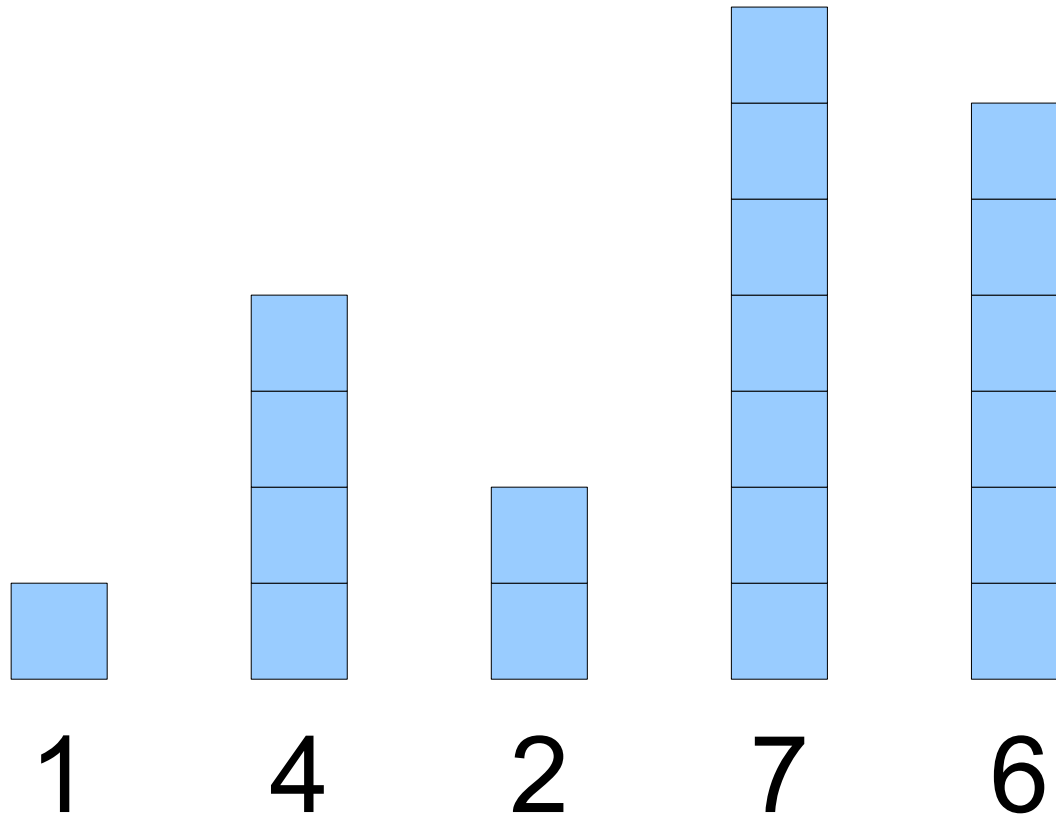
Selection Sort



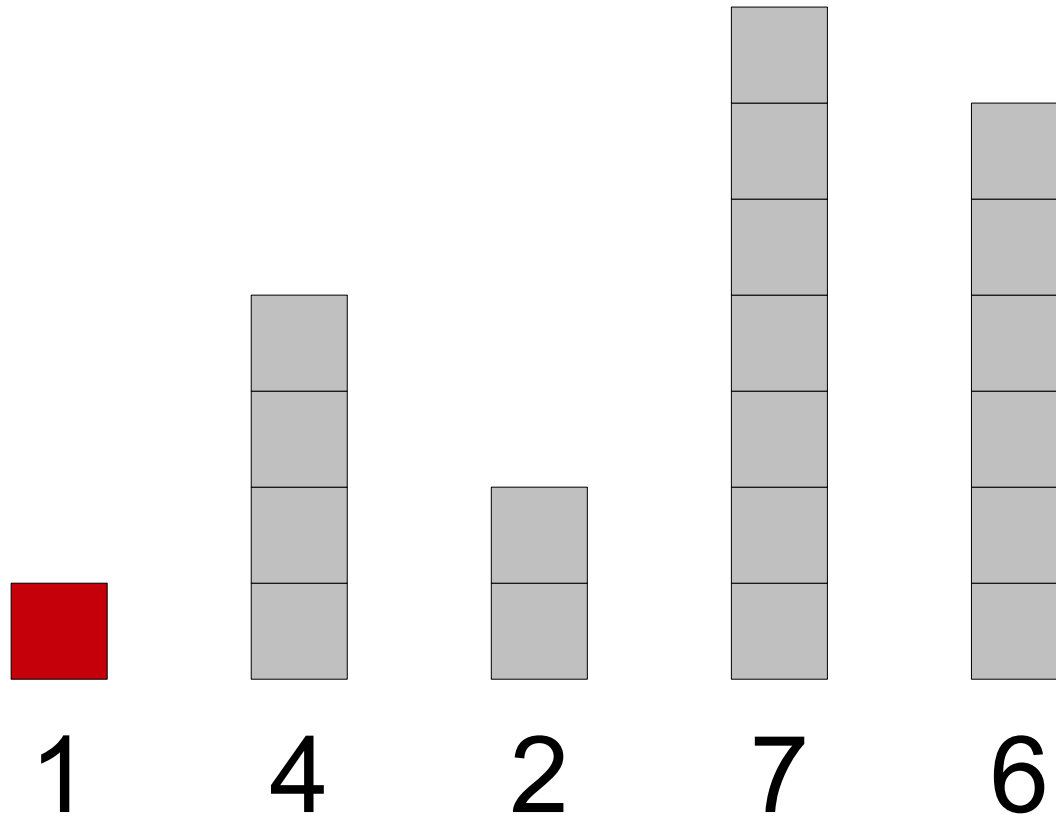
Selection Sort



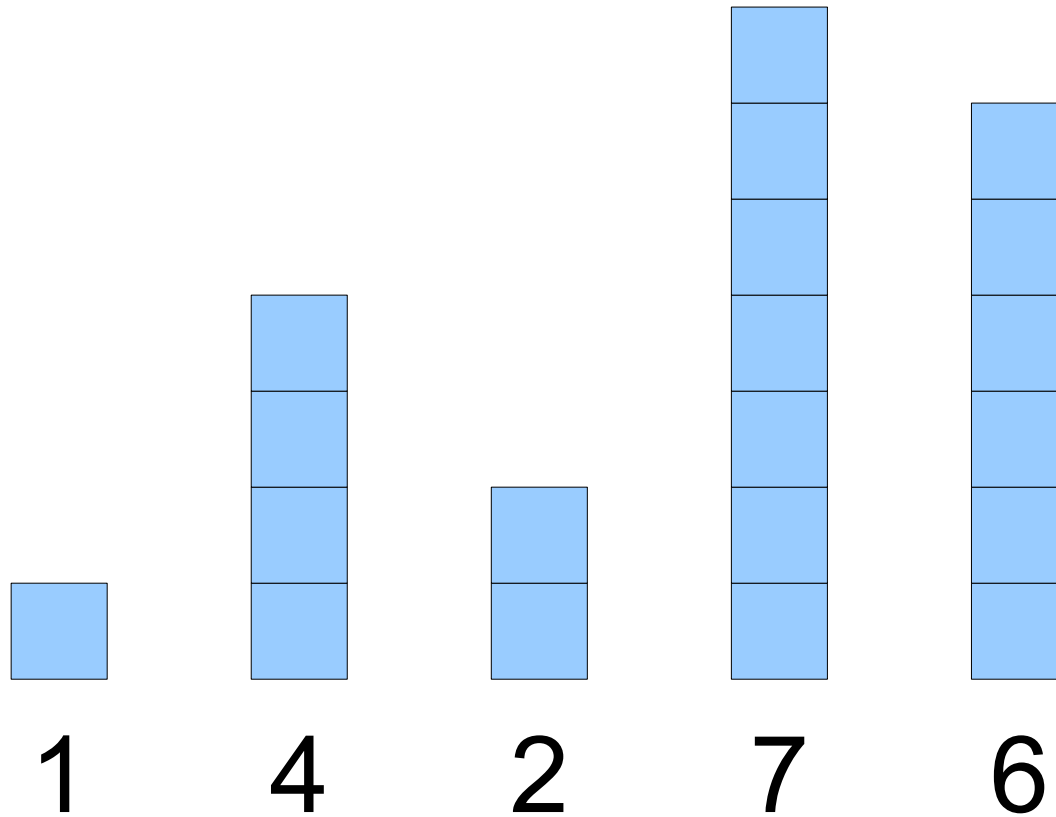
Selection Sort



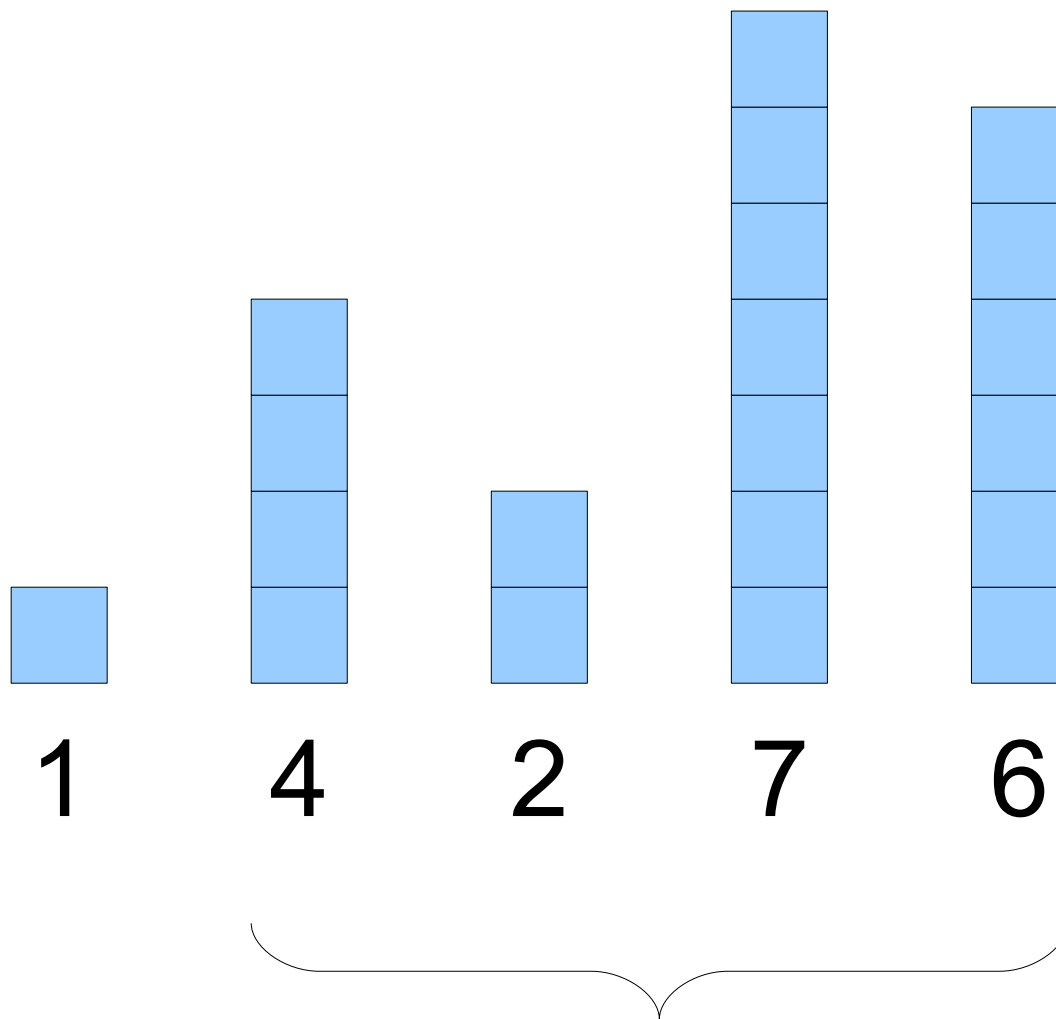
Selection Sort



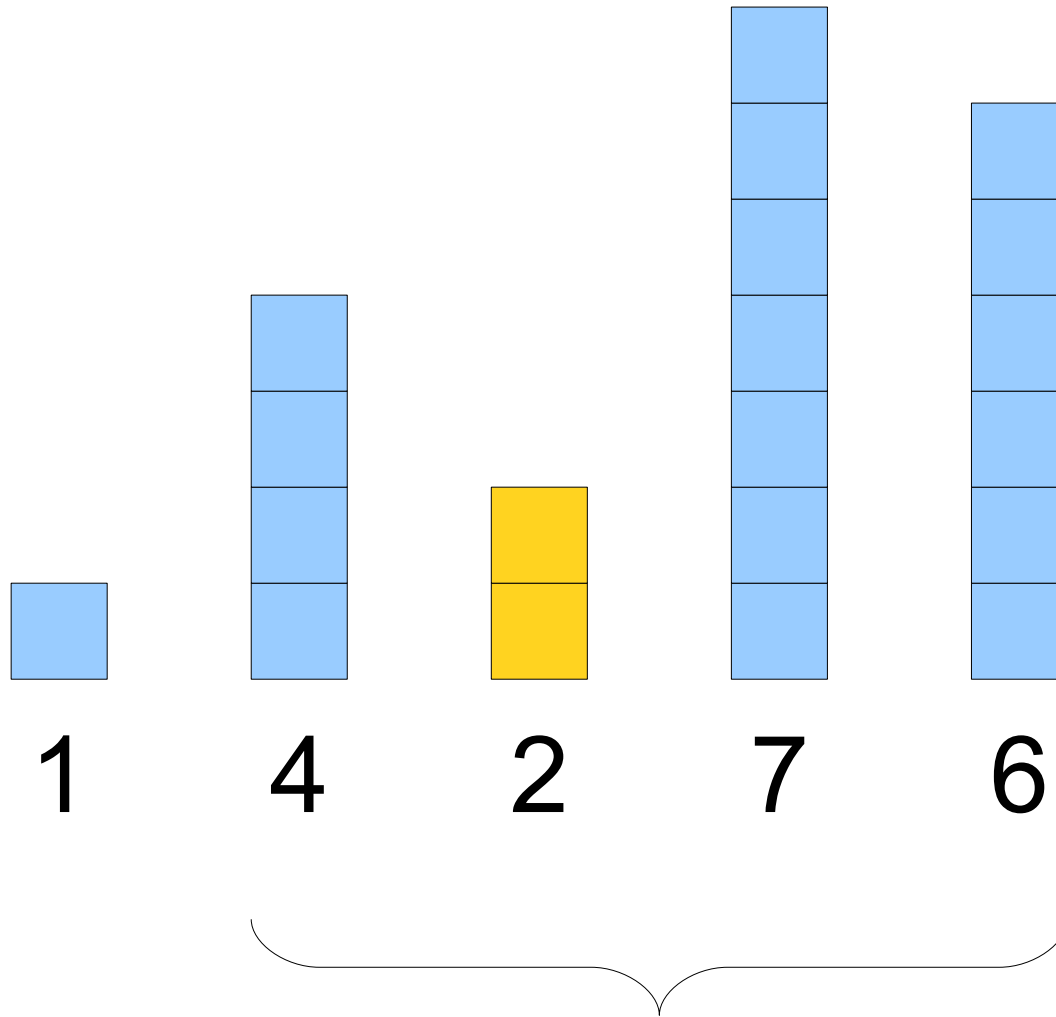
Selection Sort



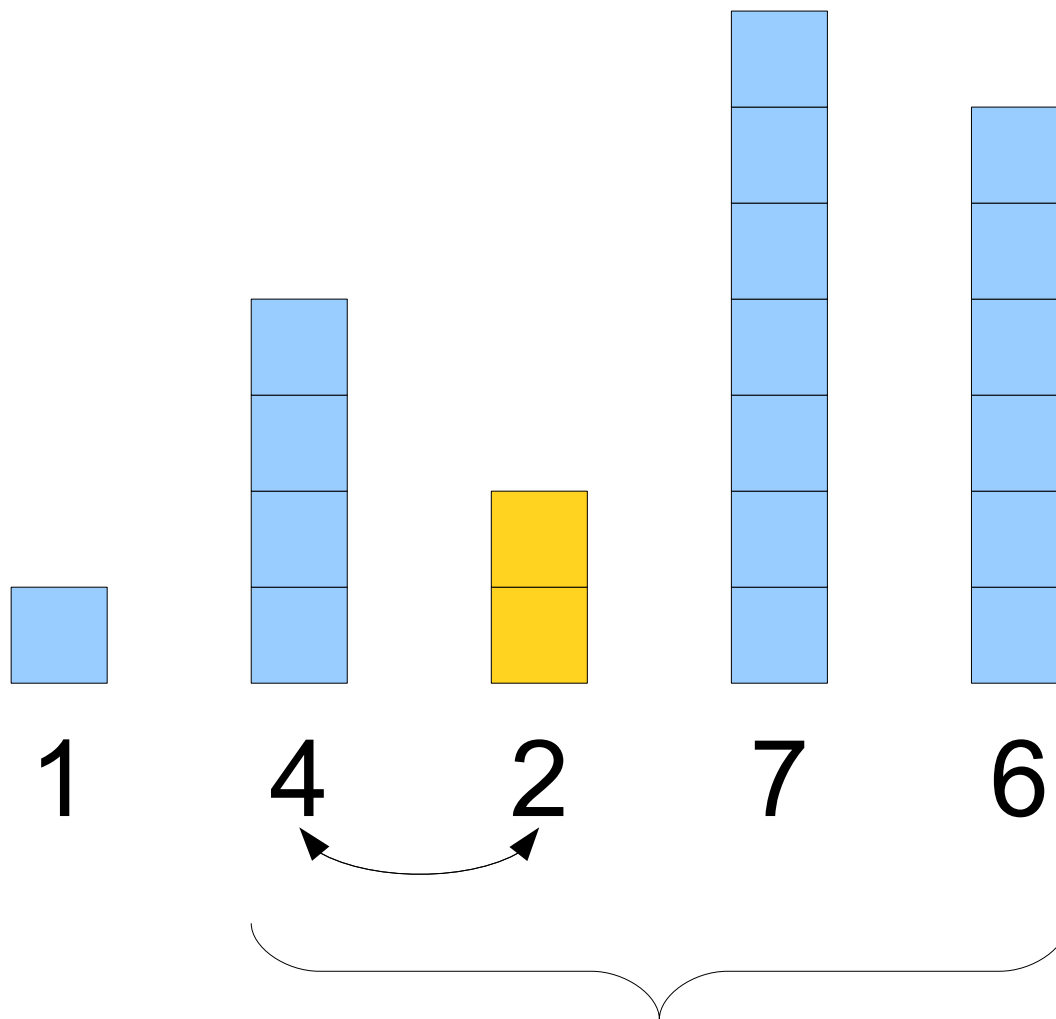
Selection Sort



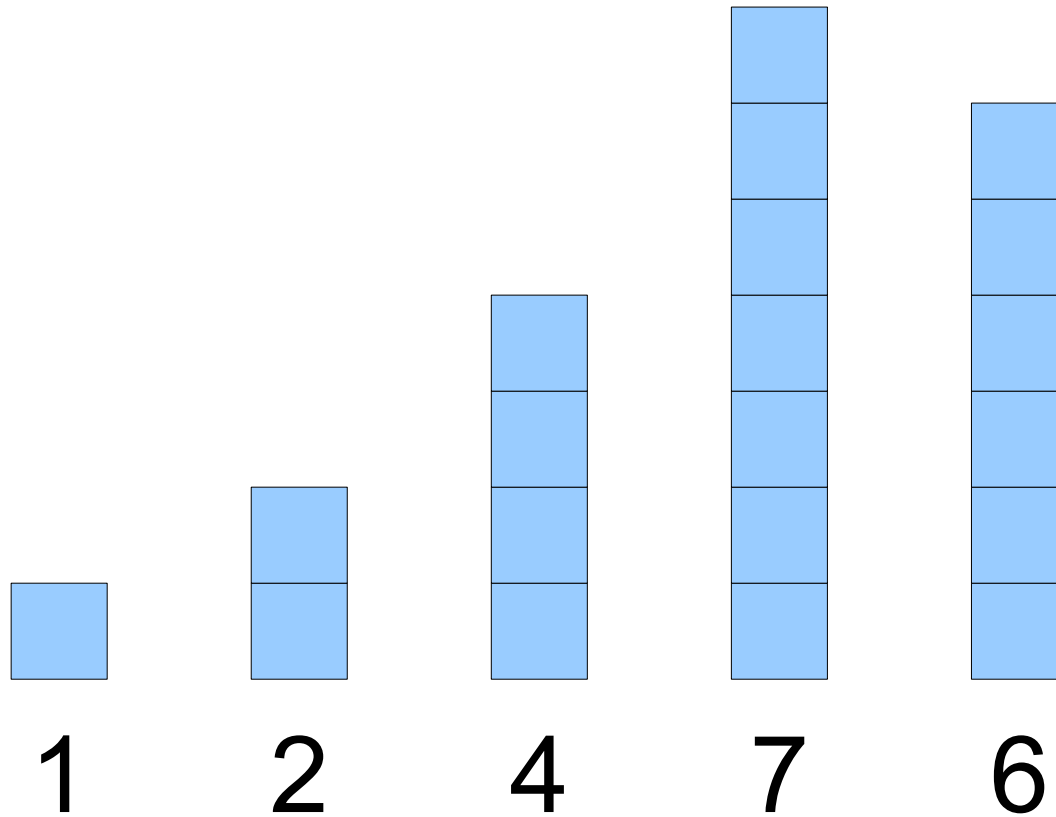
Selection Sort



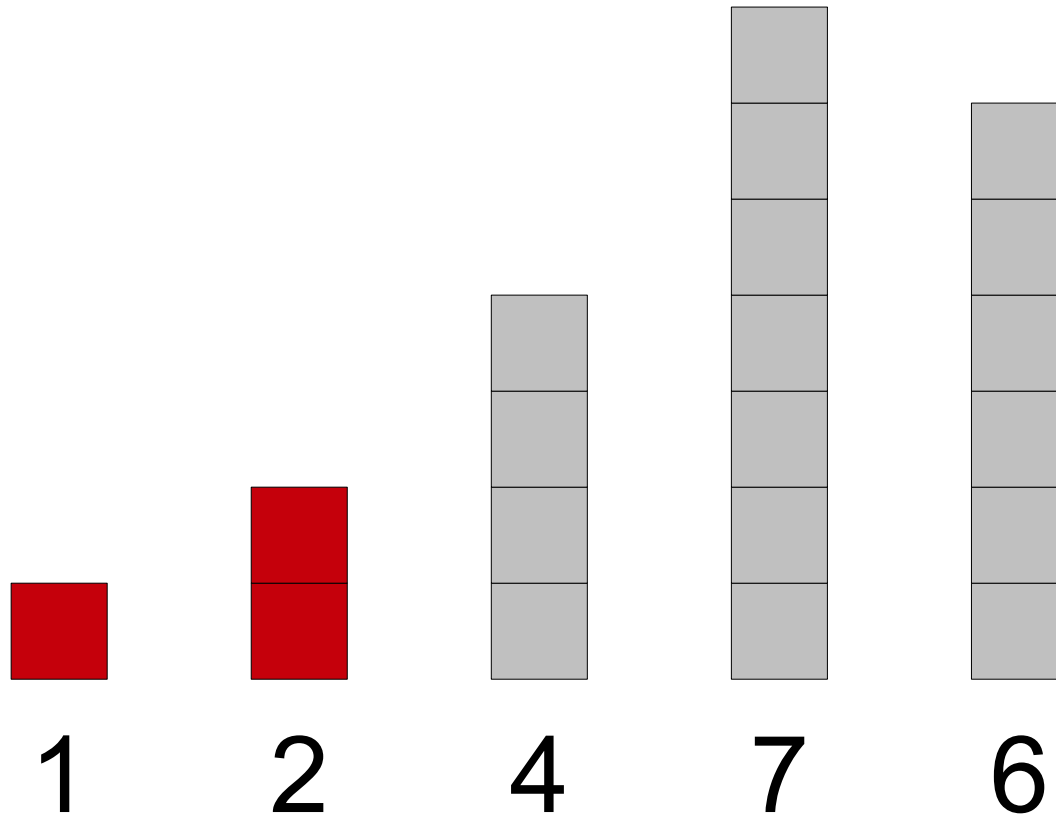
Selection Sort



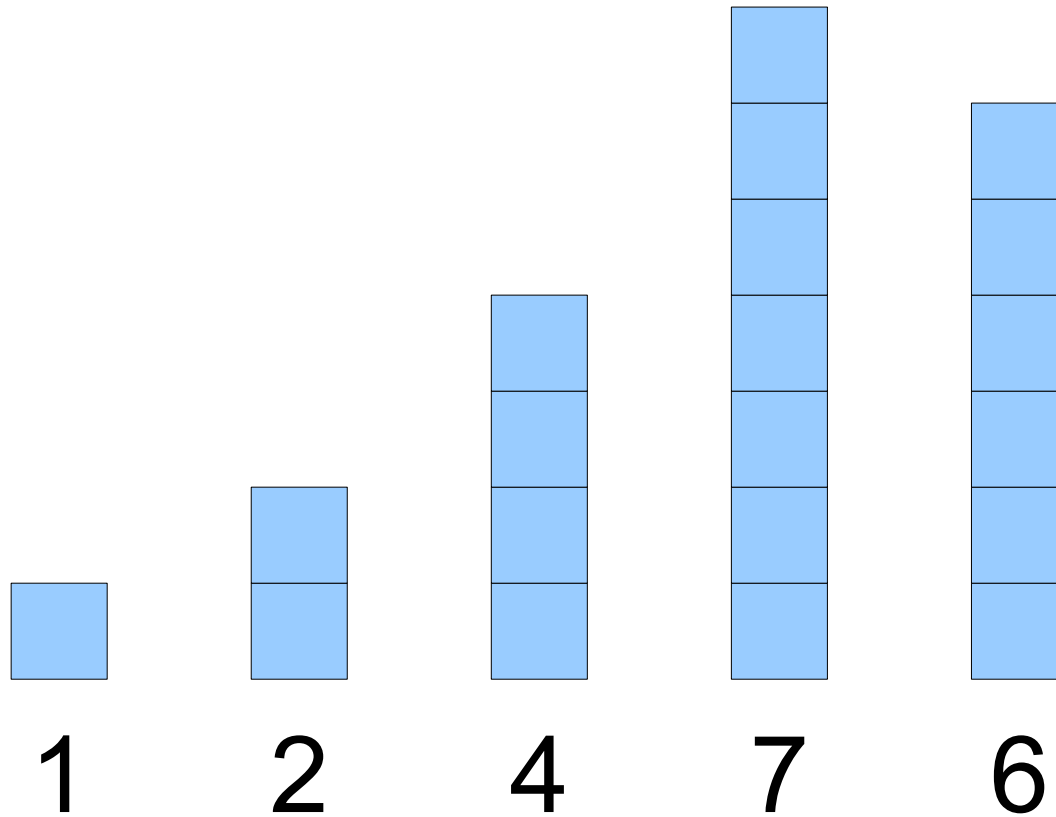
Selection Sort



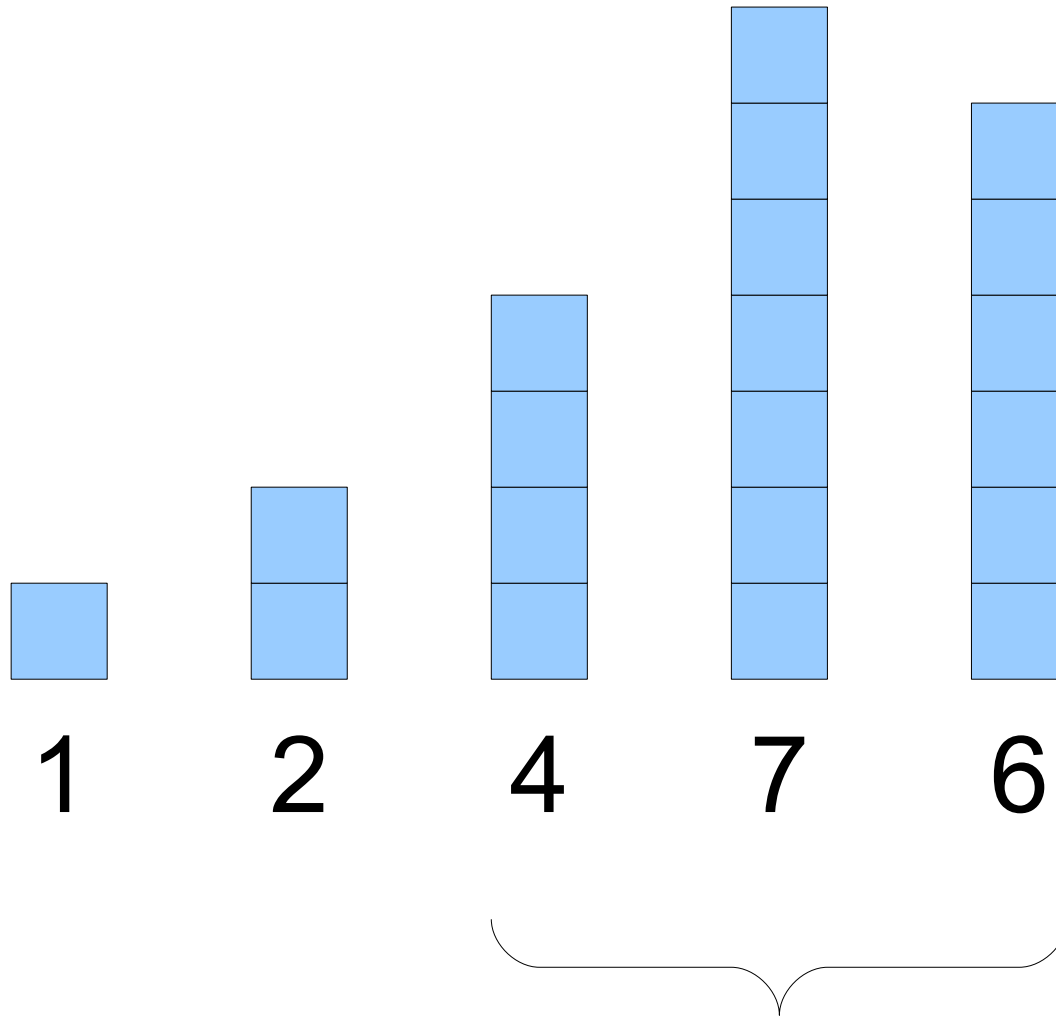
Selection Sort



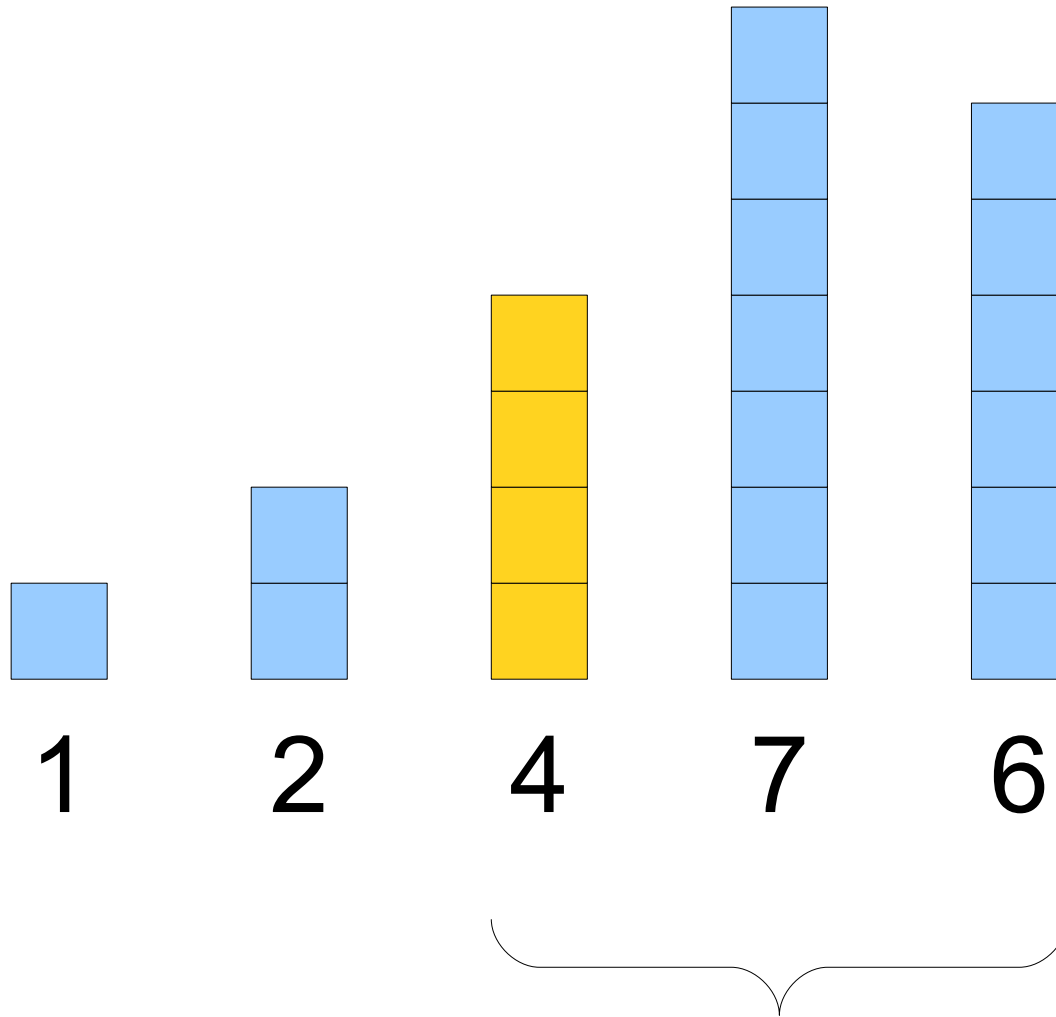
Selection Sort



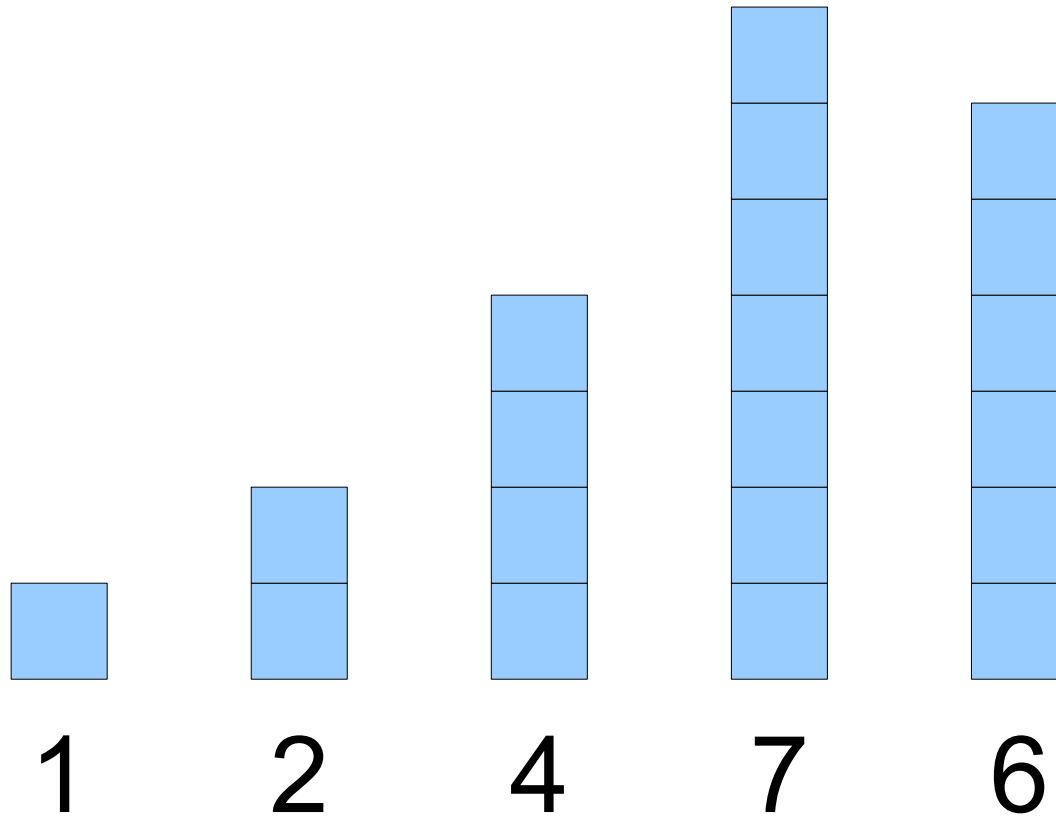
Selection Sort



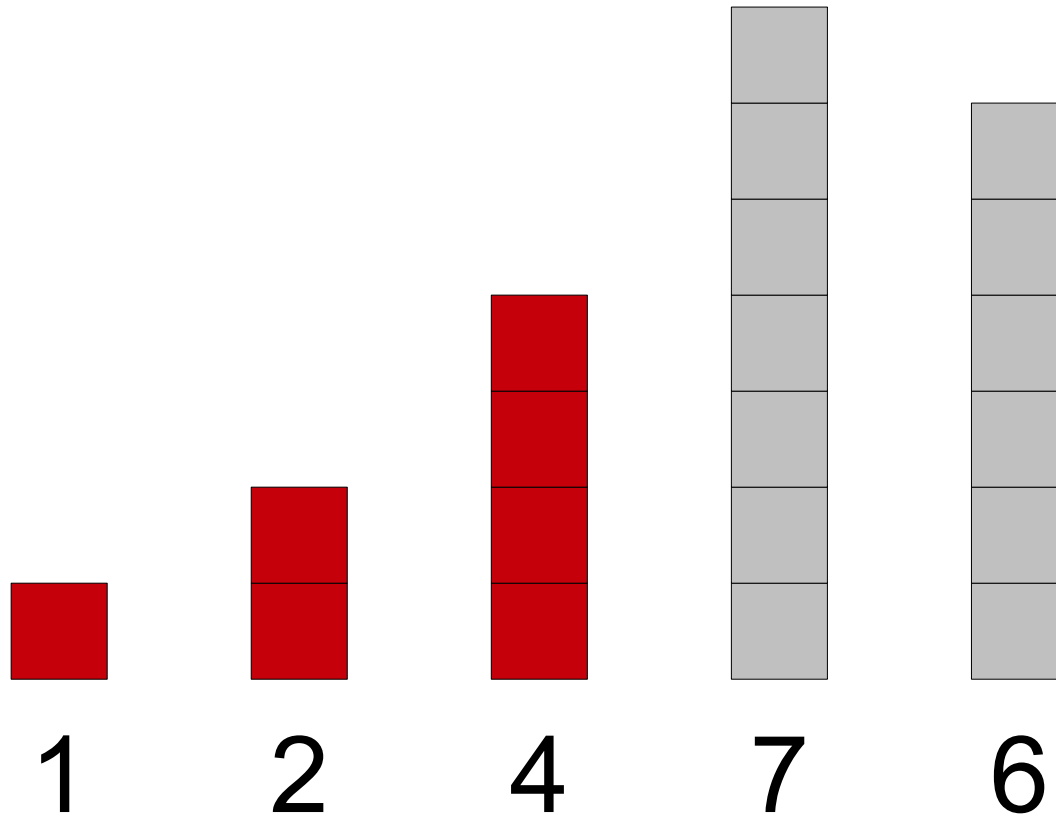
Selection Sort



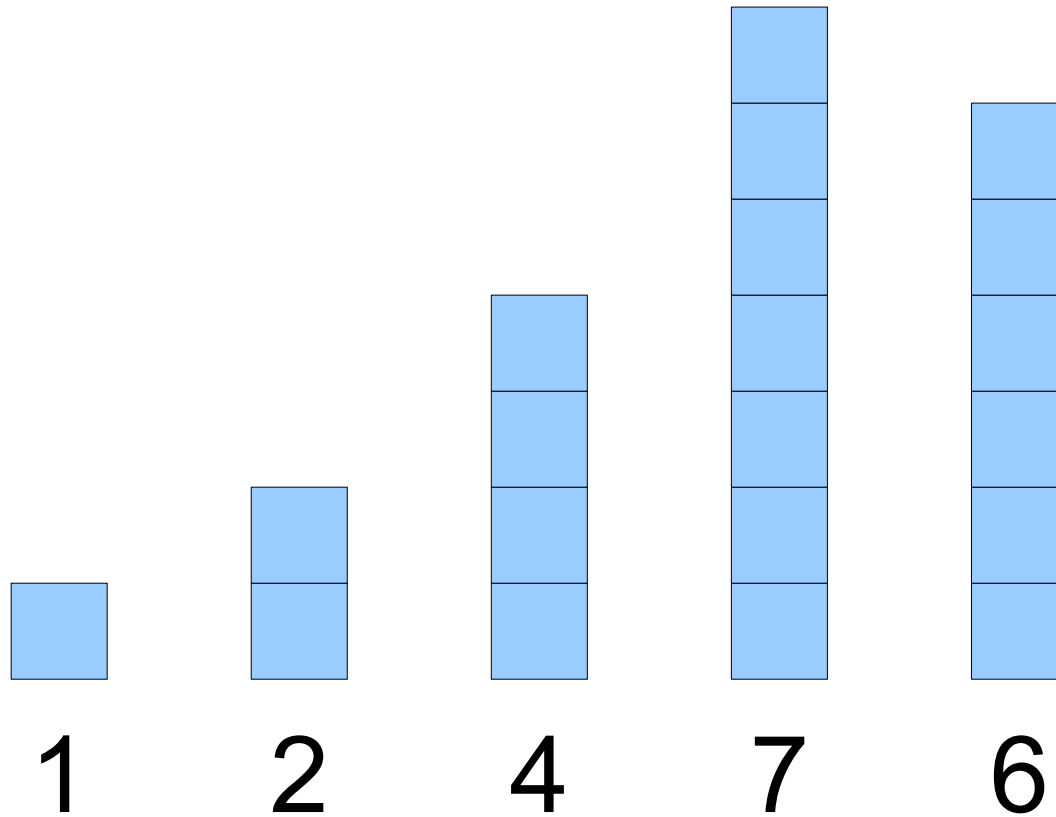
Selection Sort



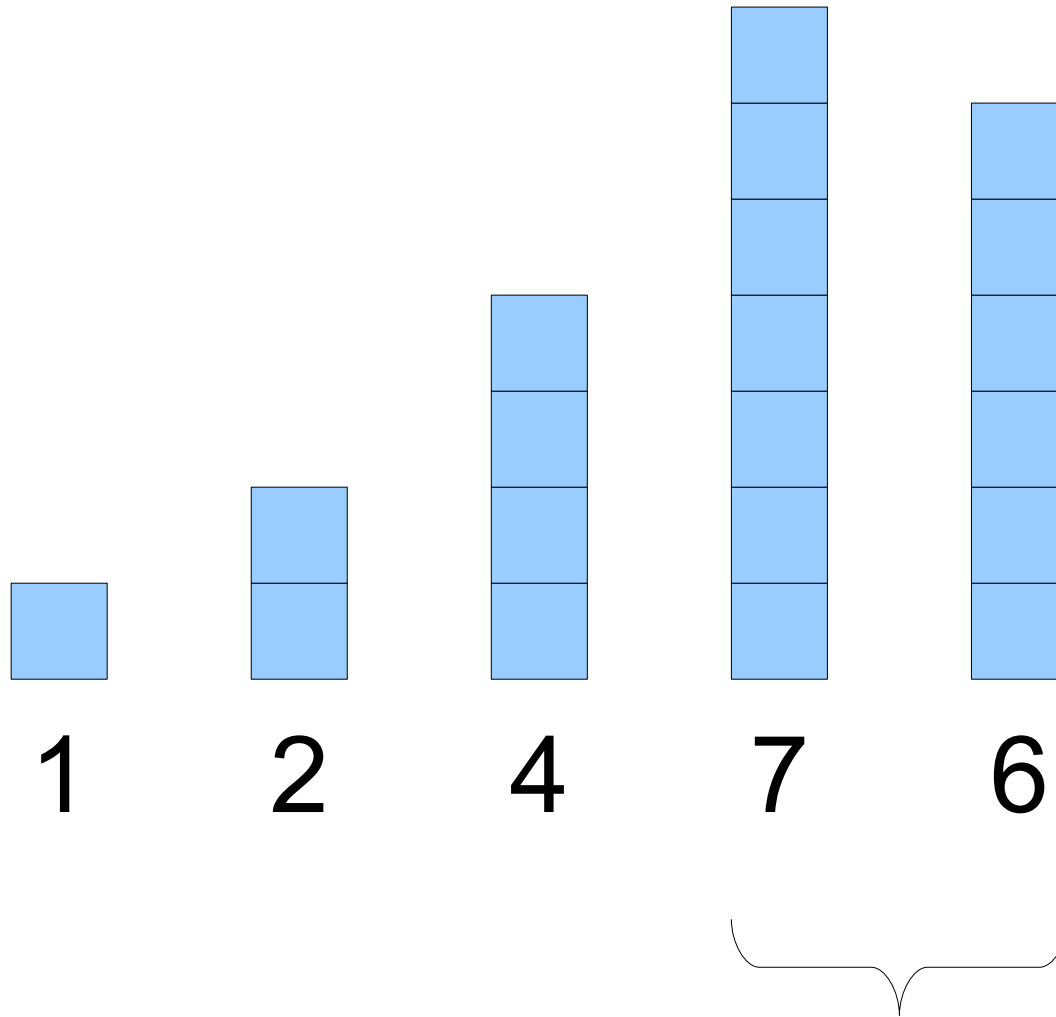
Selection Sort



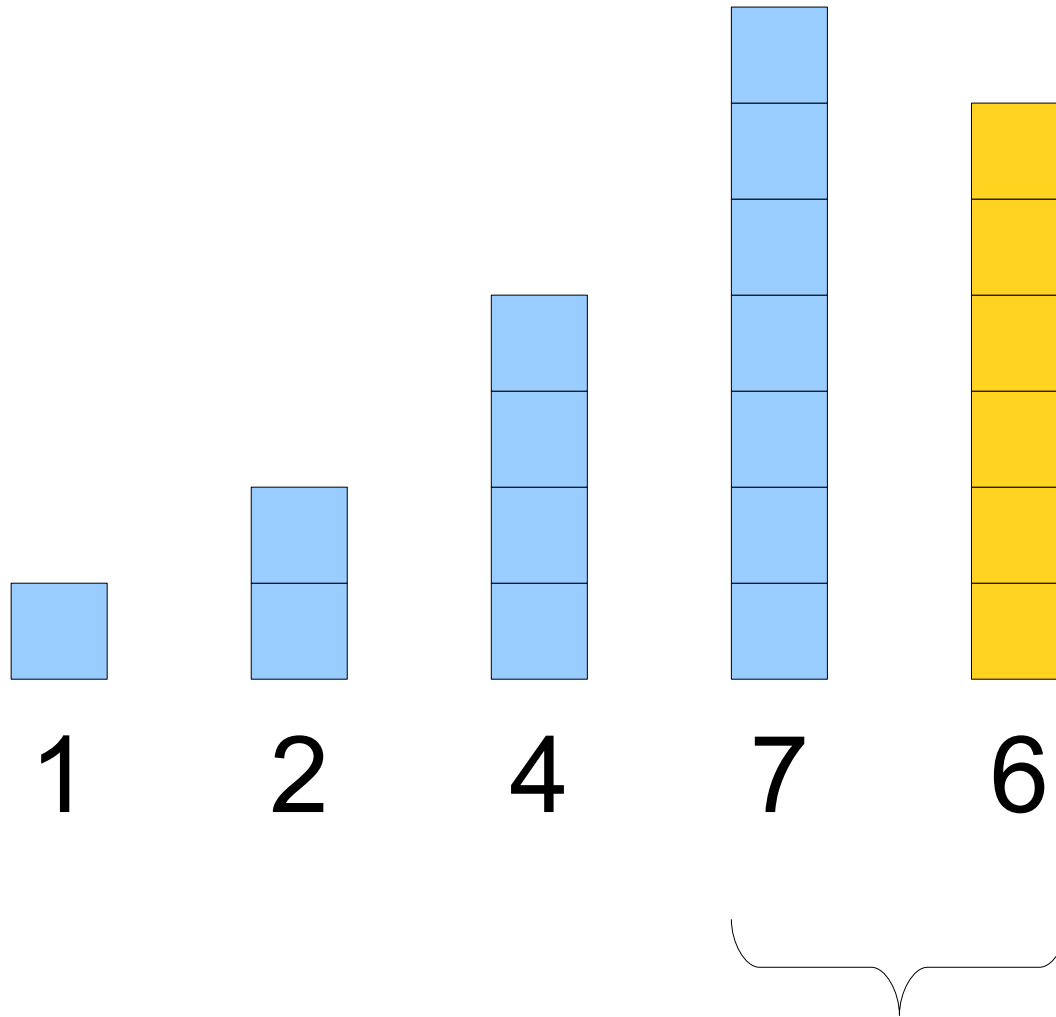
Selection Sort



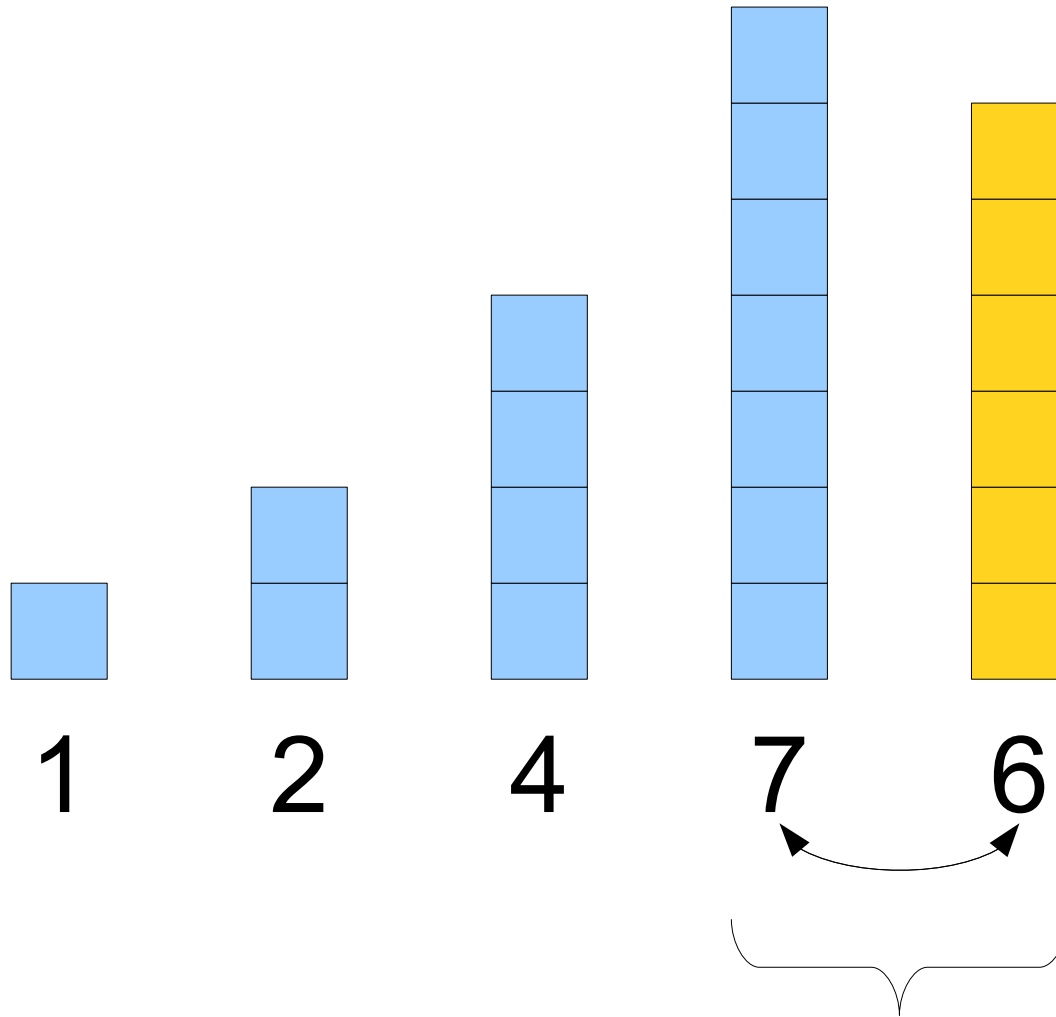
Selection Sort



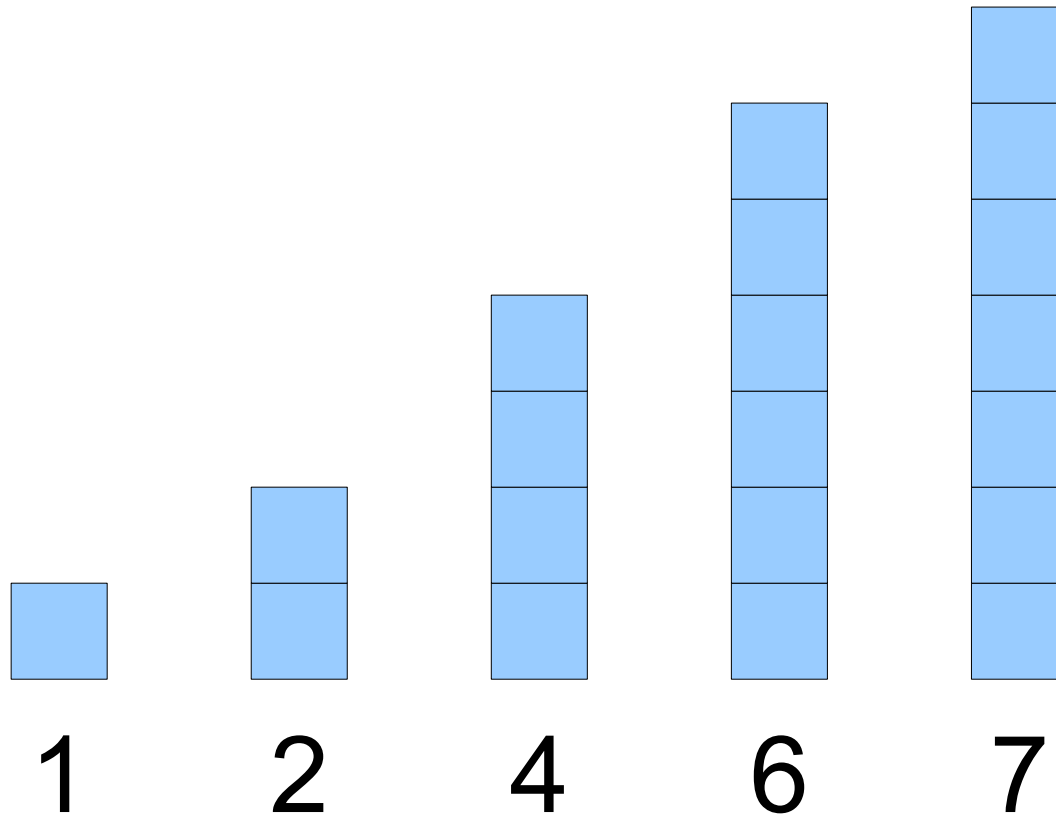
Selection Sort



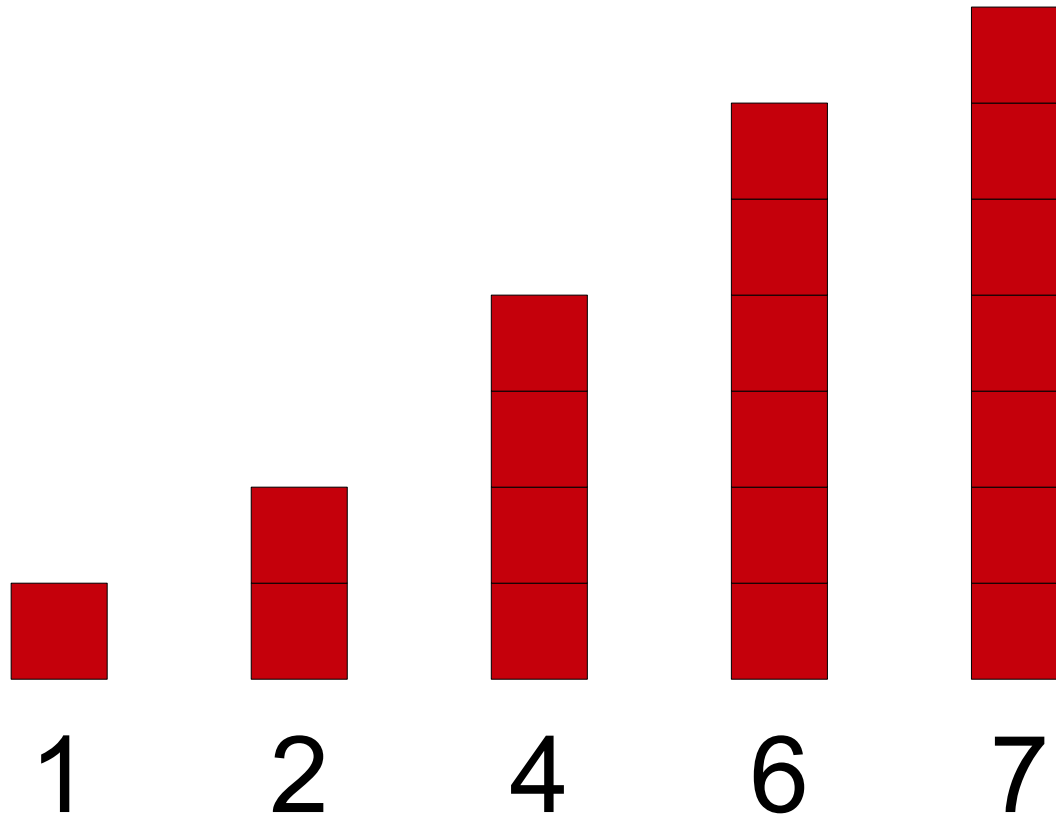
Selection Sort



Selection Sort

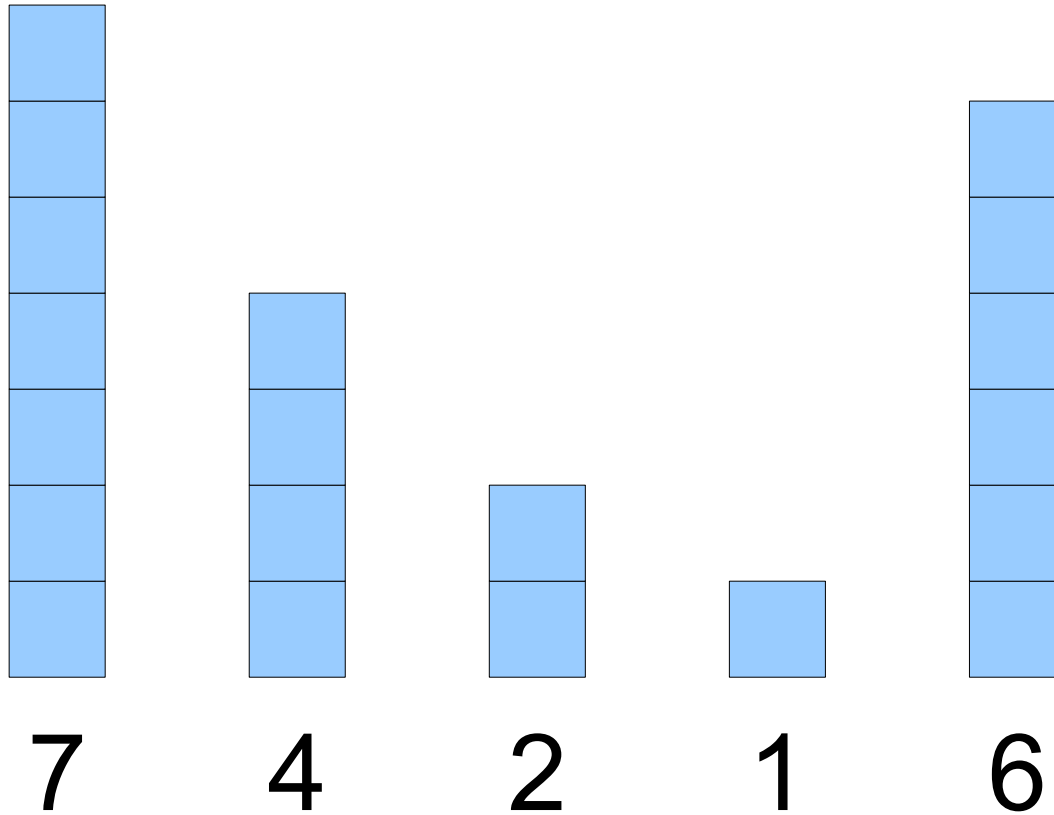


Selection Sort

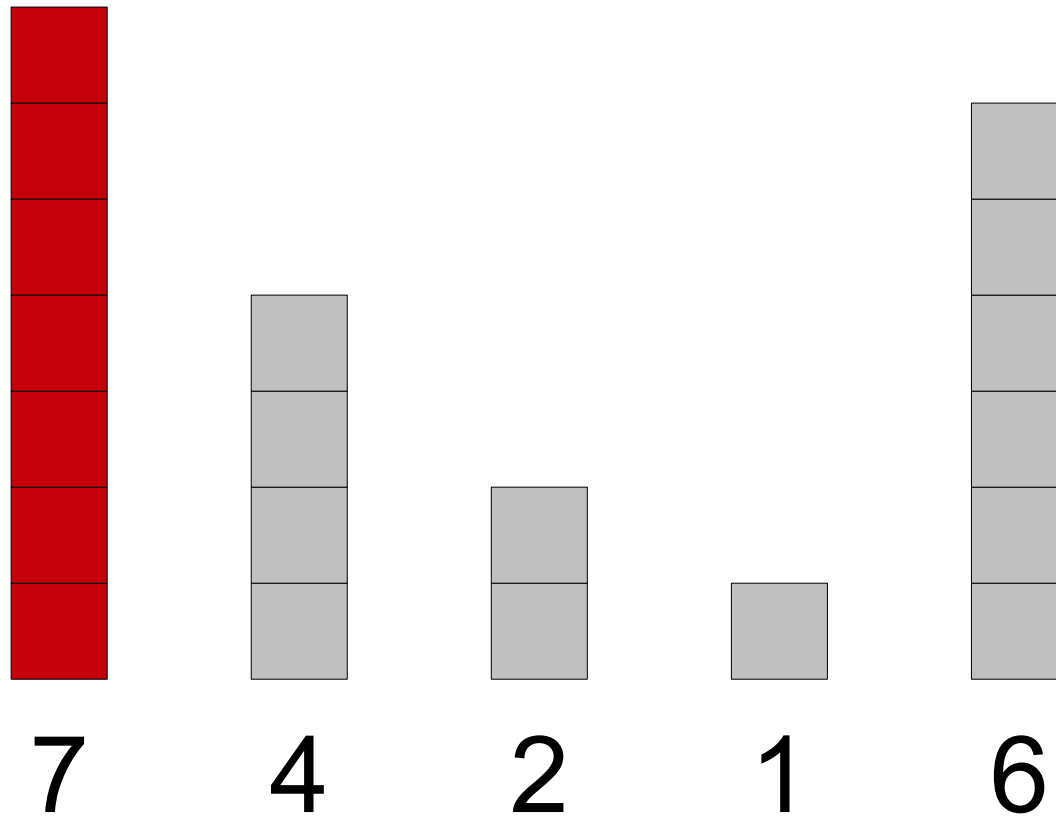


Insertion Sort

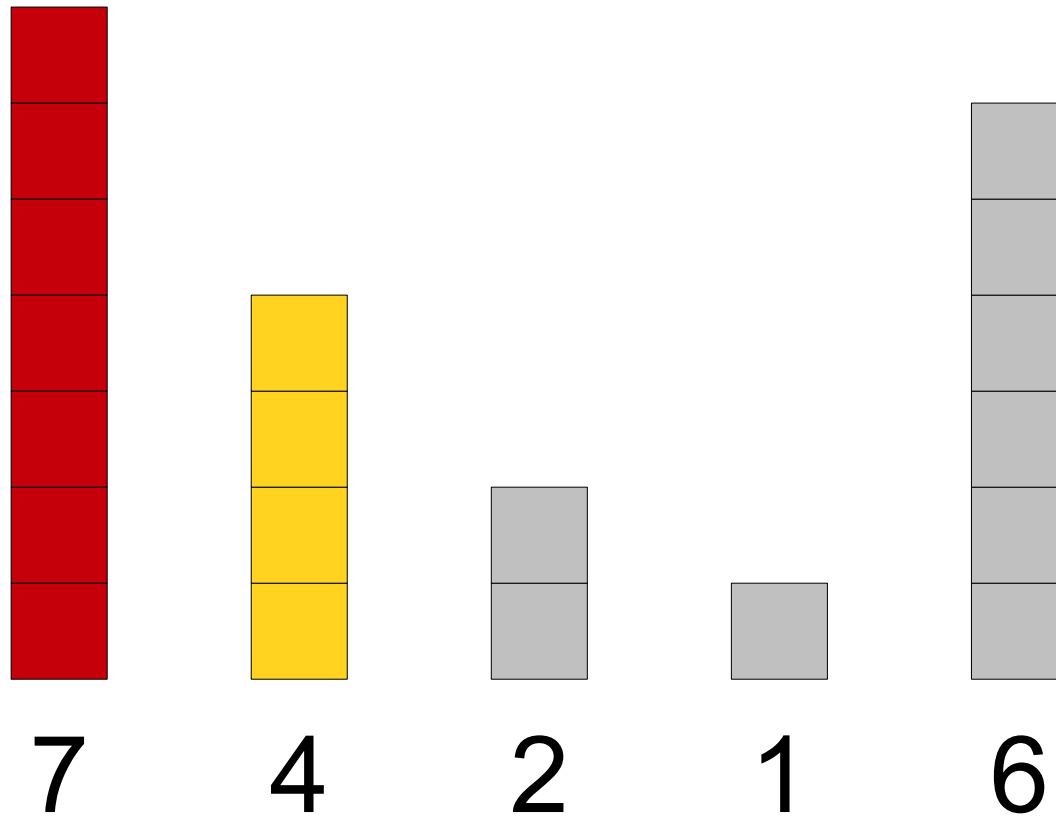
Insertion Sort



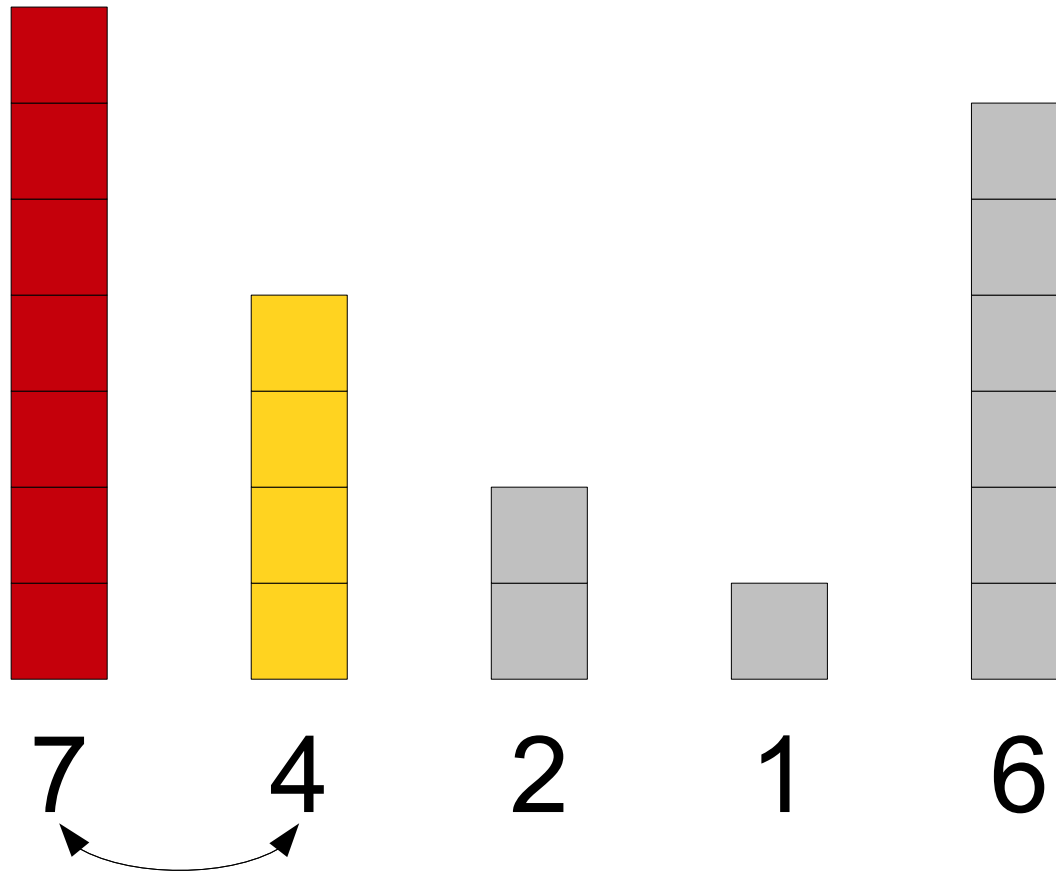
Insertion Sort



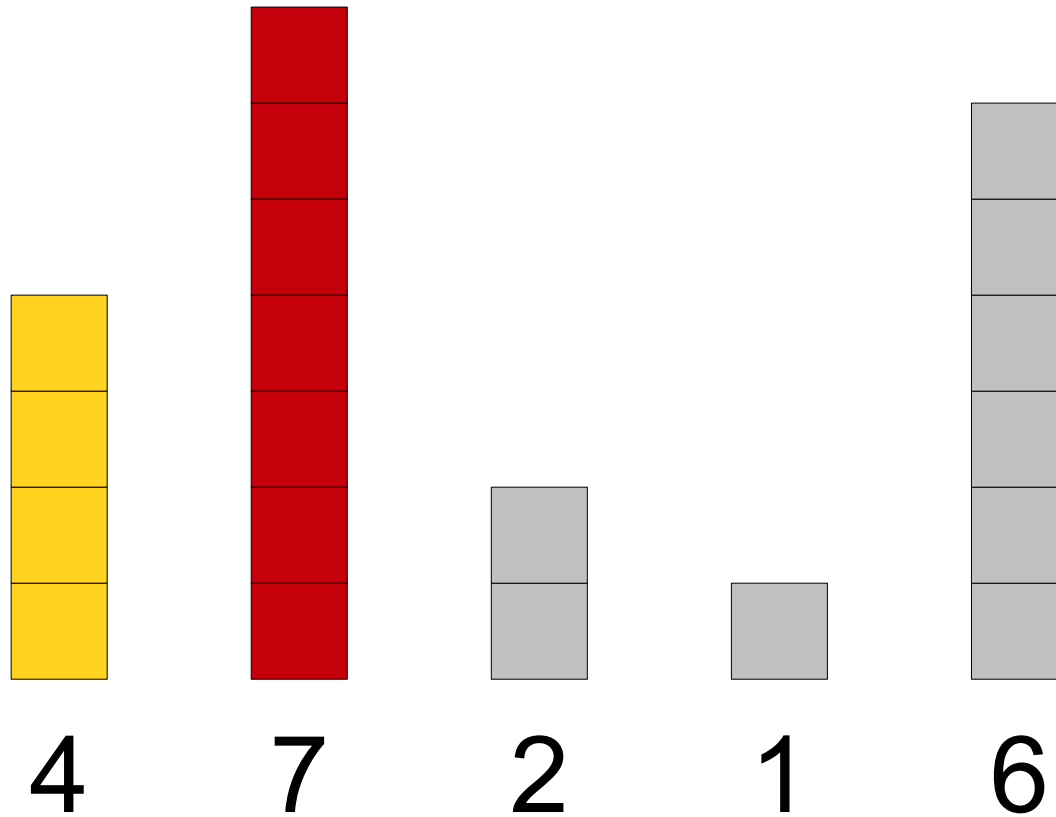
Insertion Sort



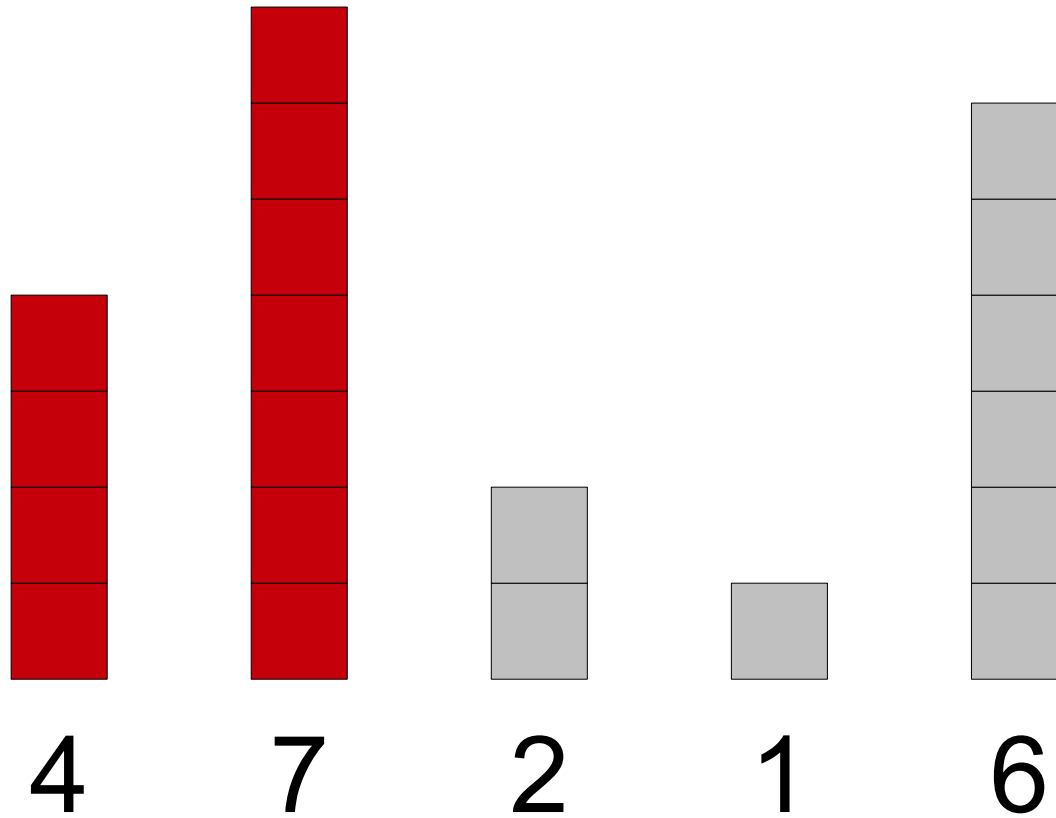
Insertion Sort



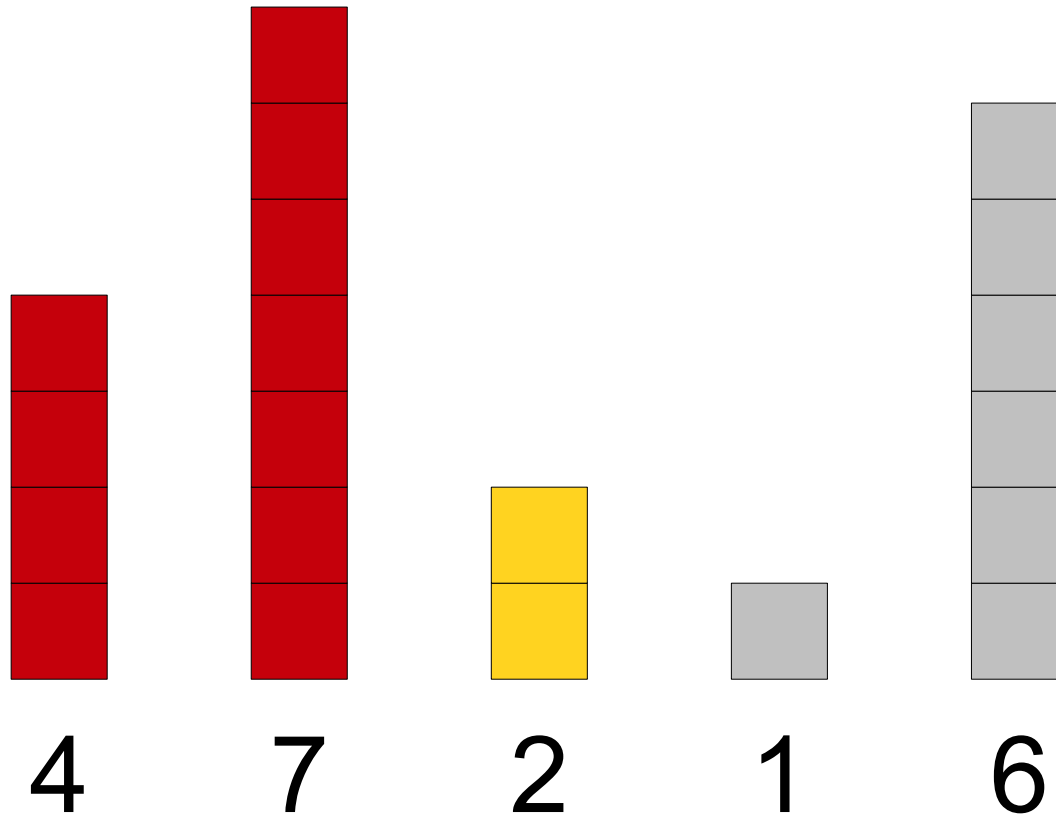
Insertion Sort



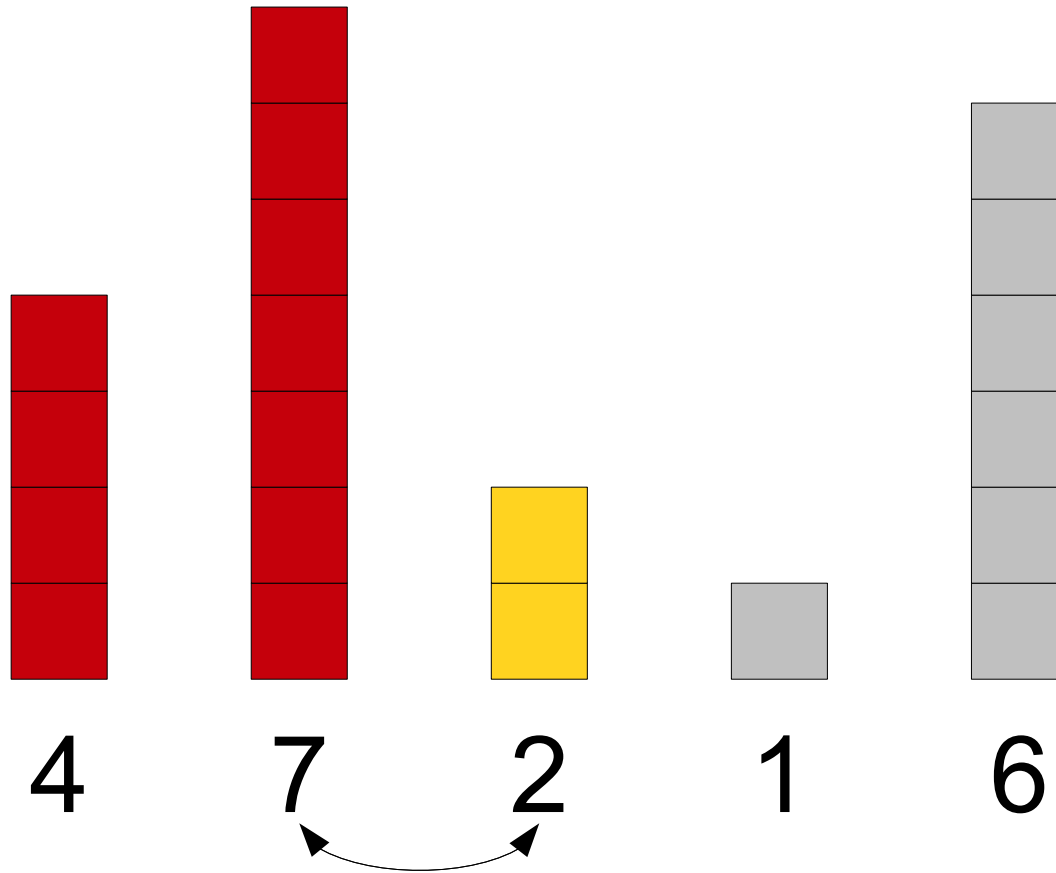
Insertion Sort



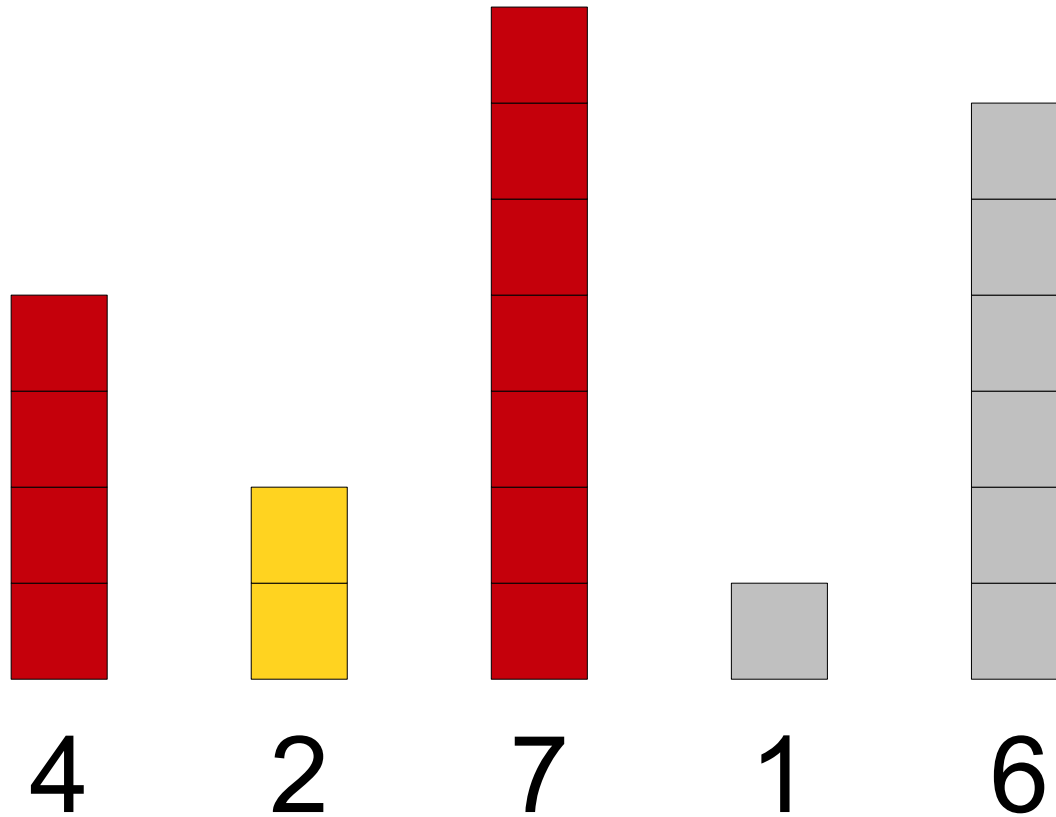
Insertion Sort



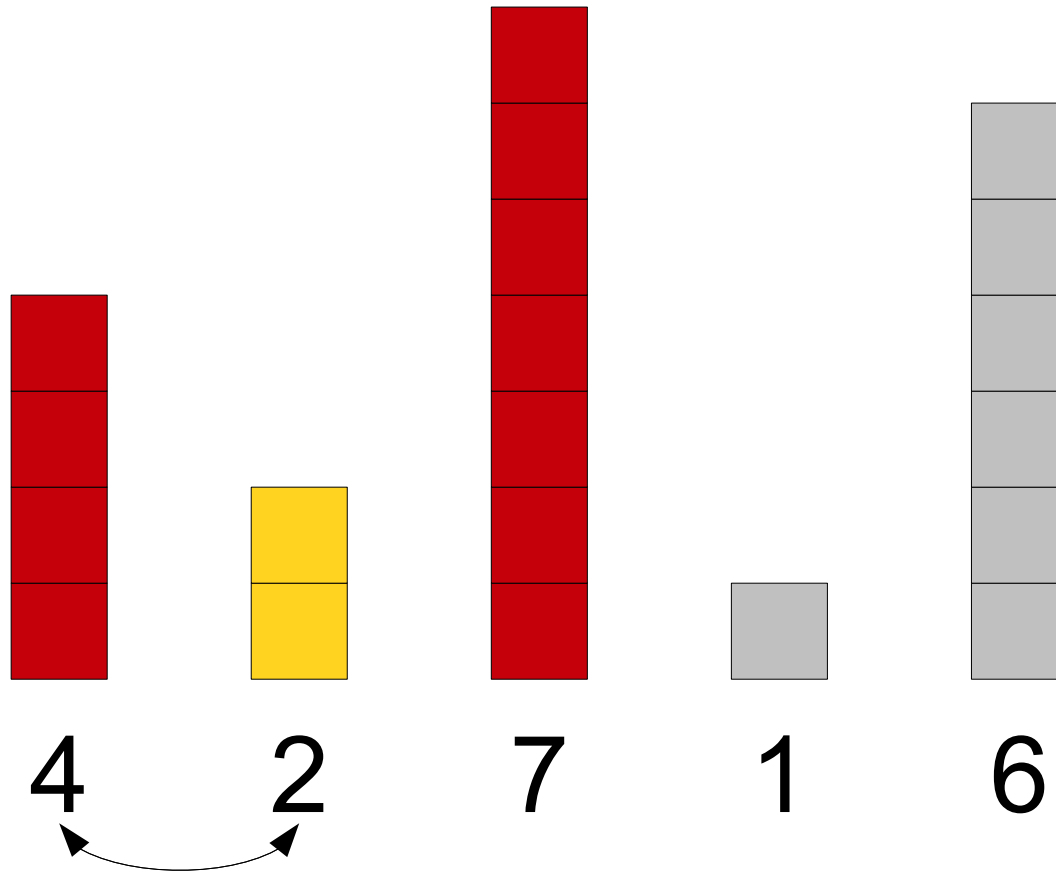
Insertion Sort



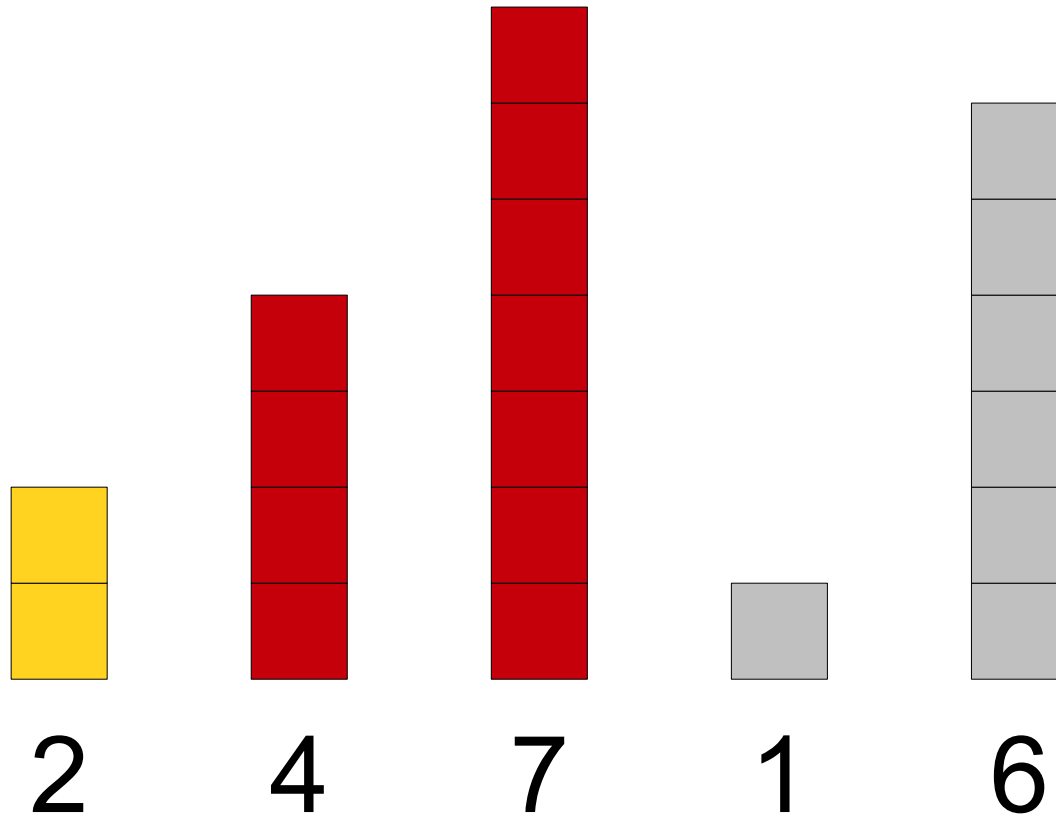
Insertion Sort



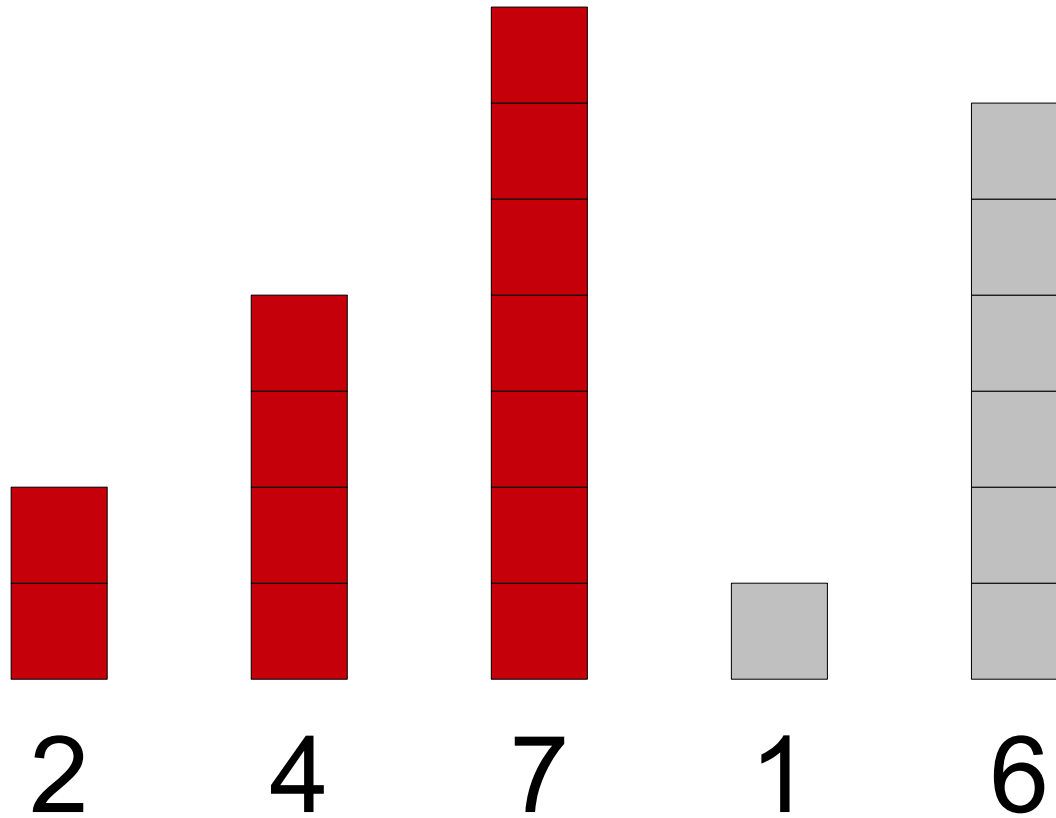
Insertion Sort



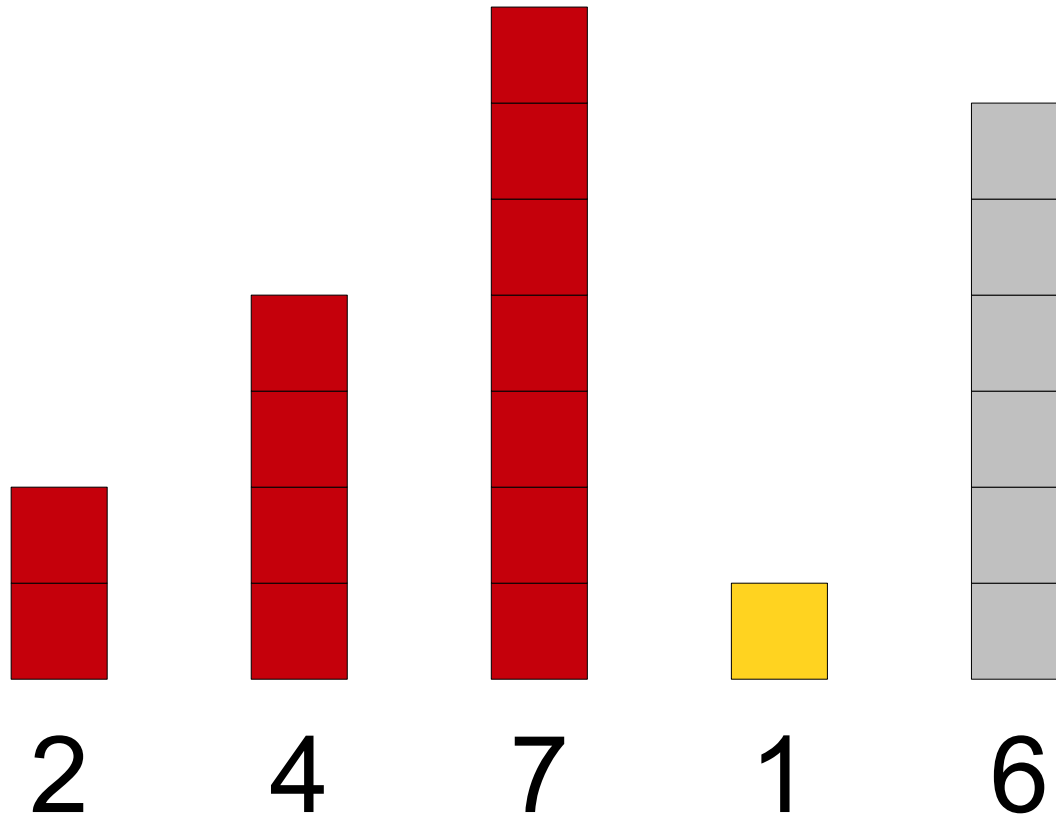
Insertion Sort



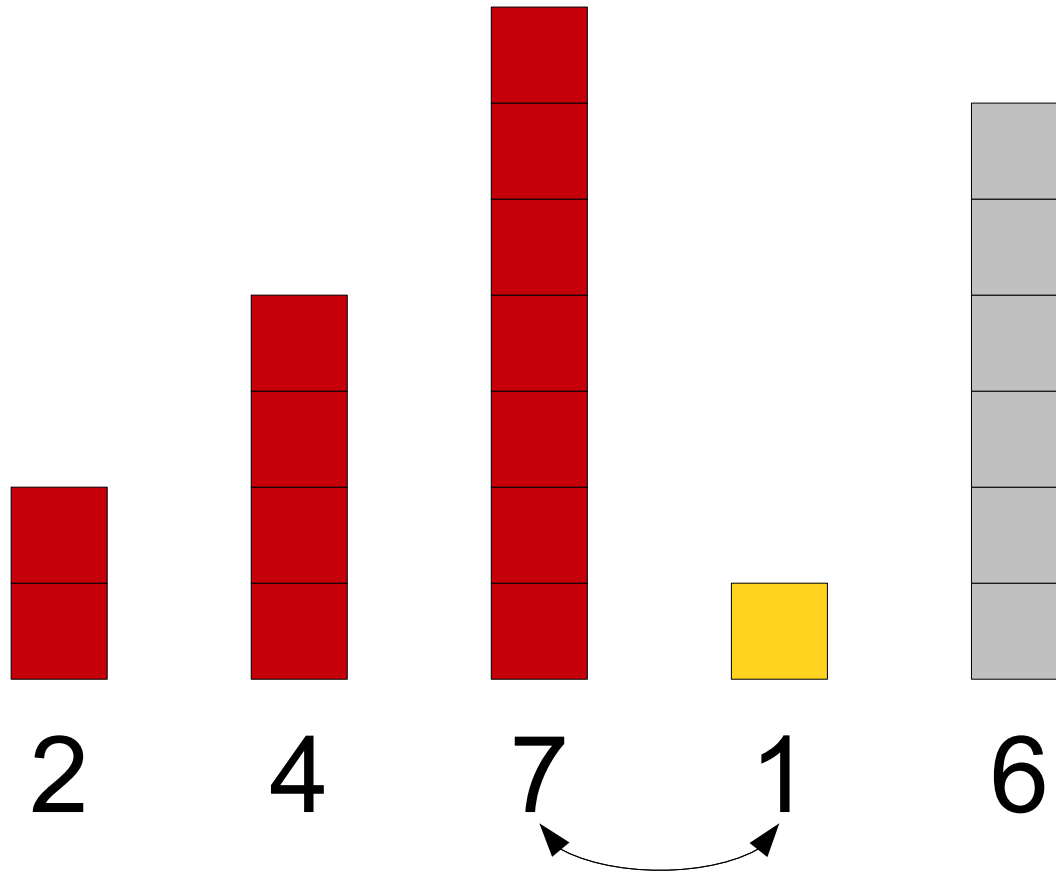
Insertion Sort



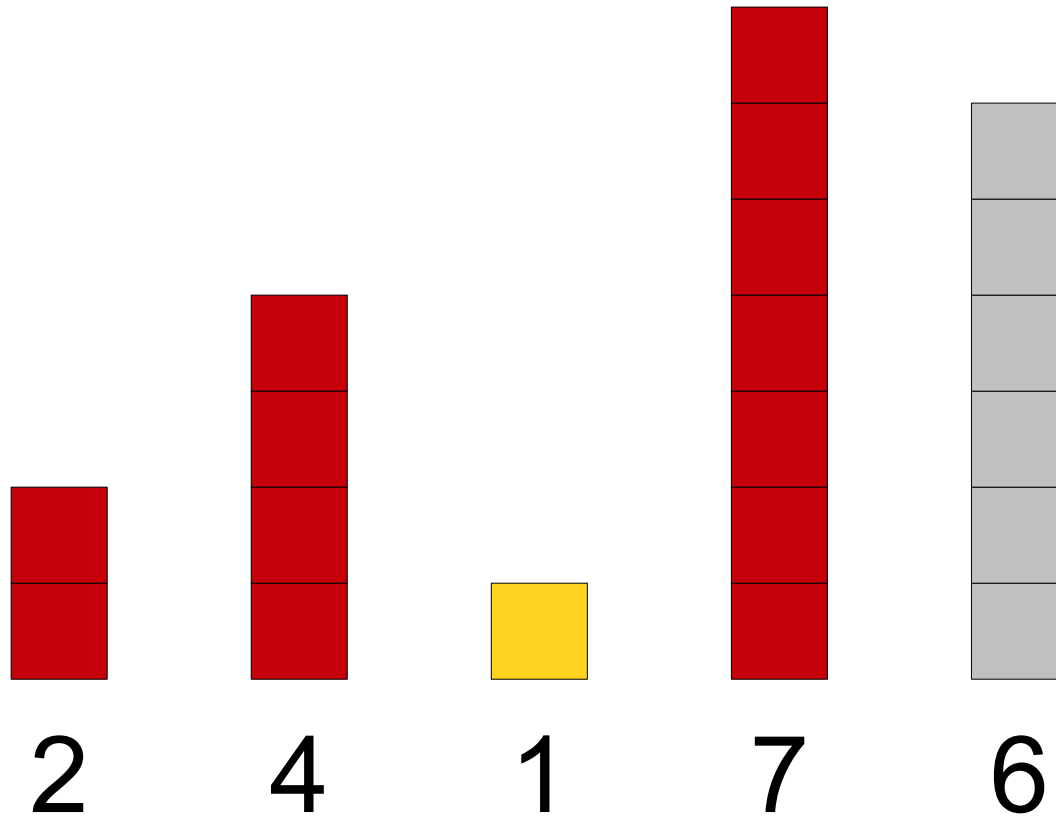
Insertion Sort



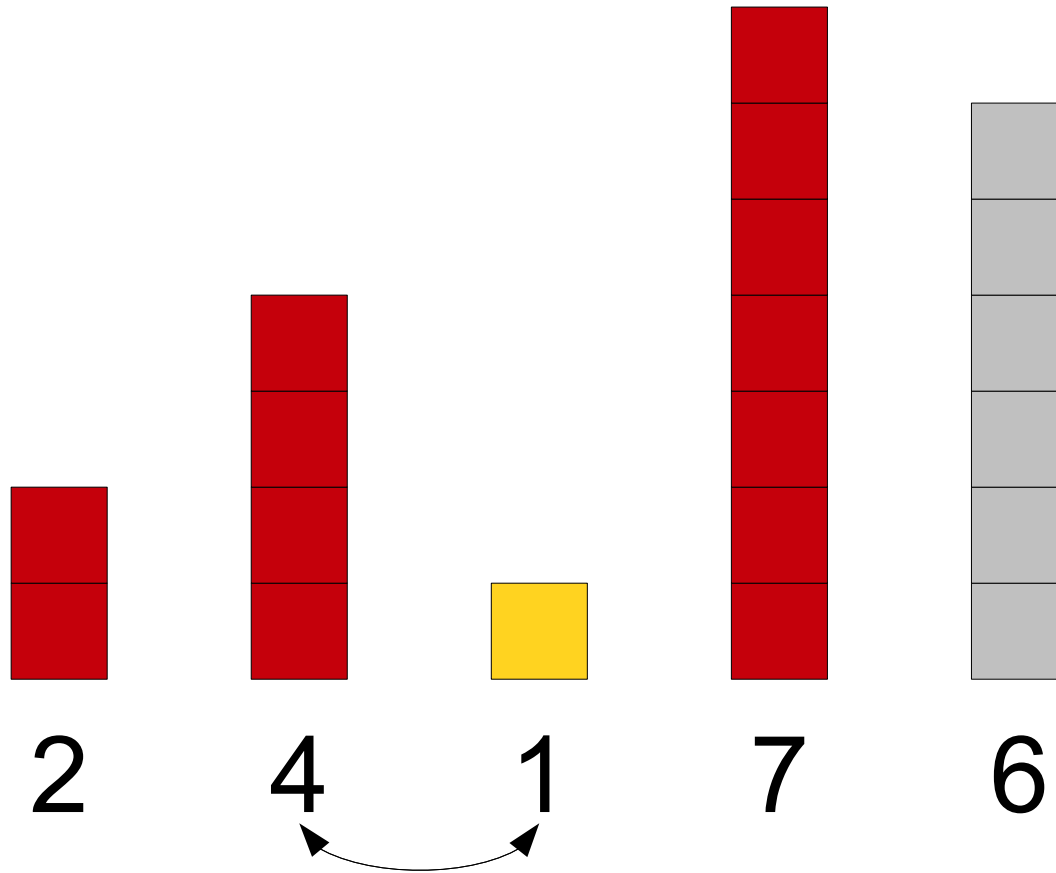
Insertion Sort



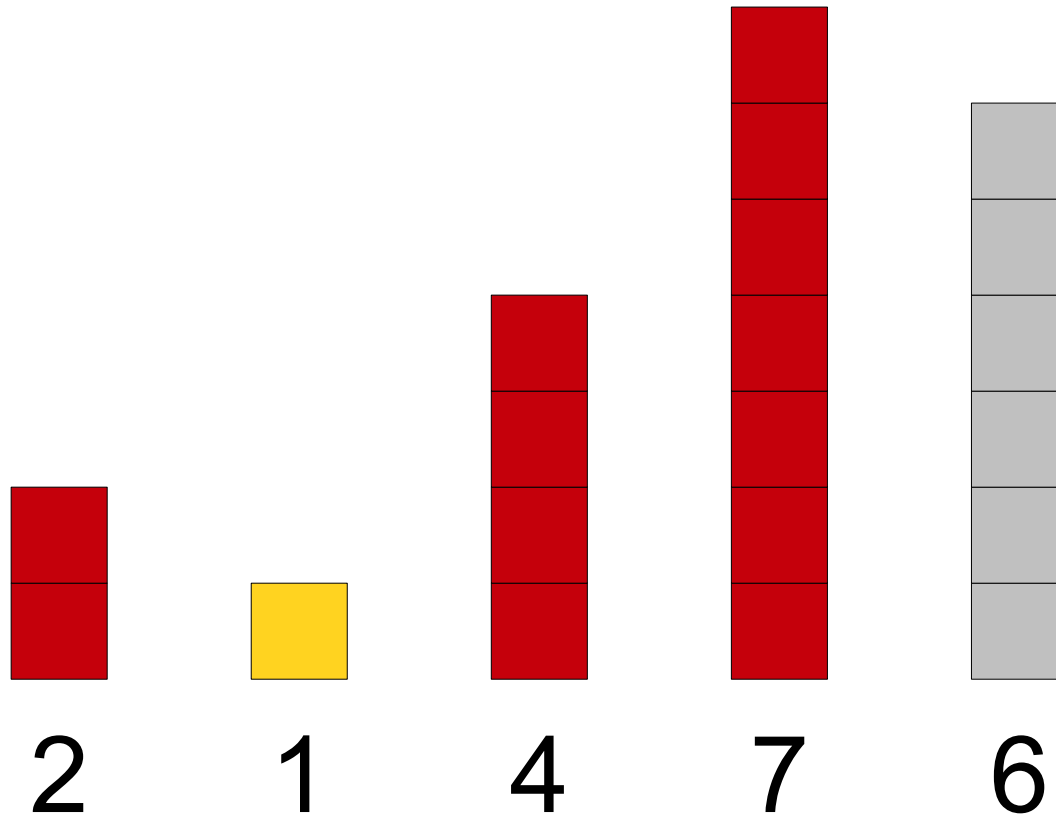
Insertion Sort



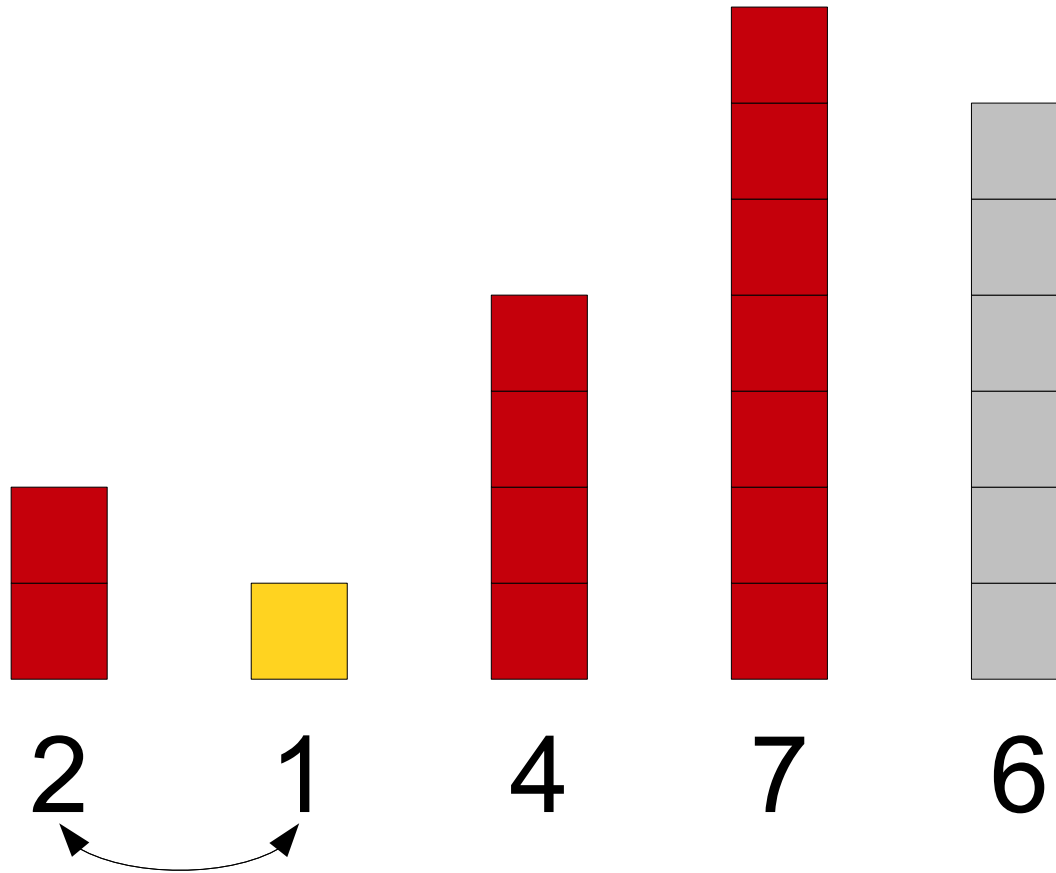
Insertion Sort



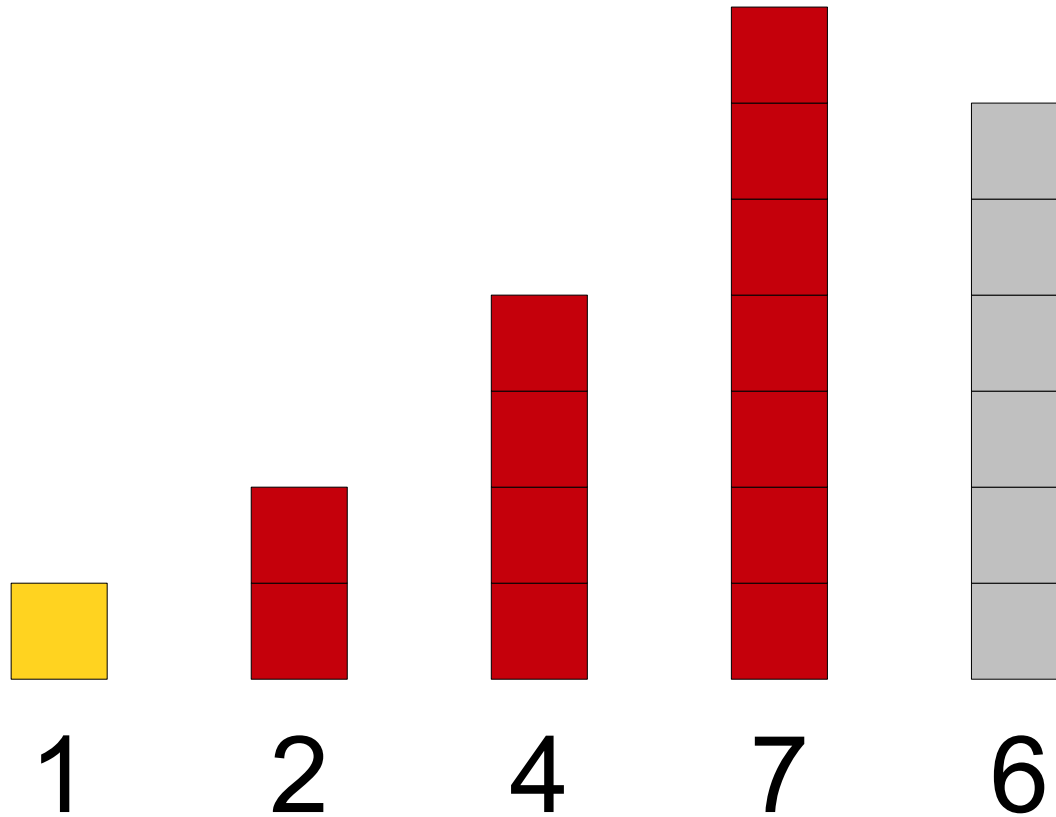
Insertion Sort



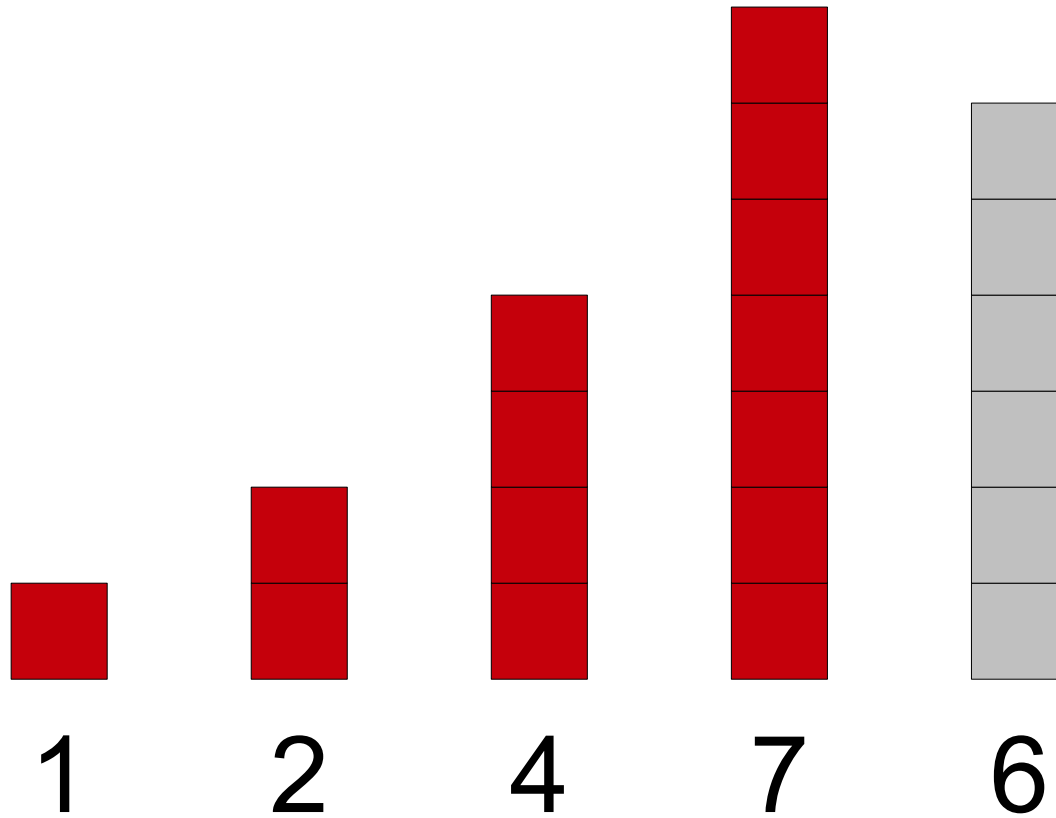
Insertion Sort



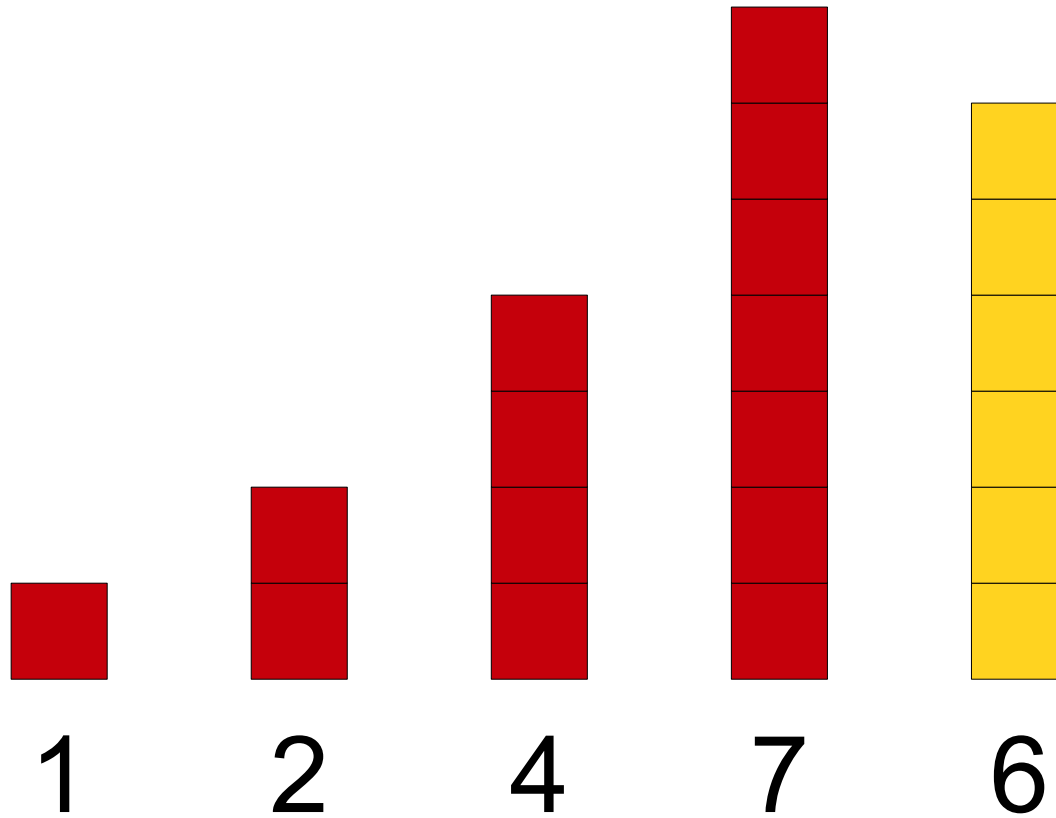
Insertion Sort



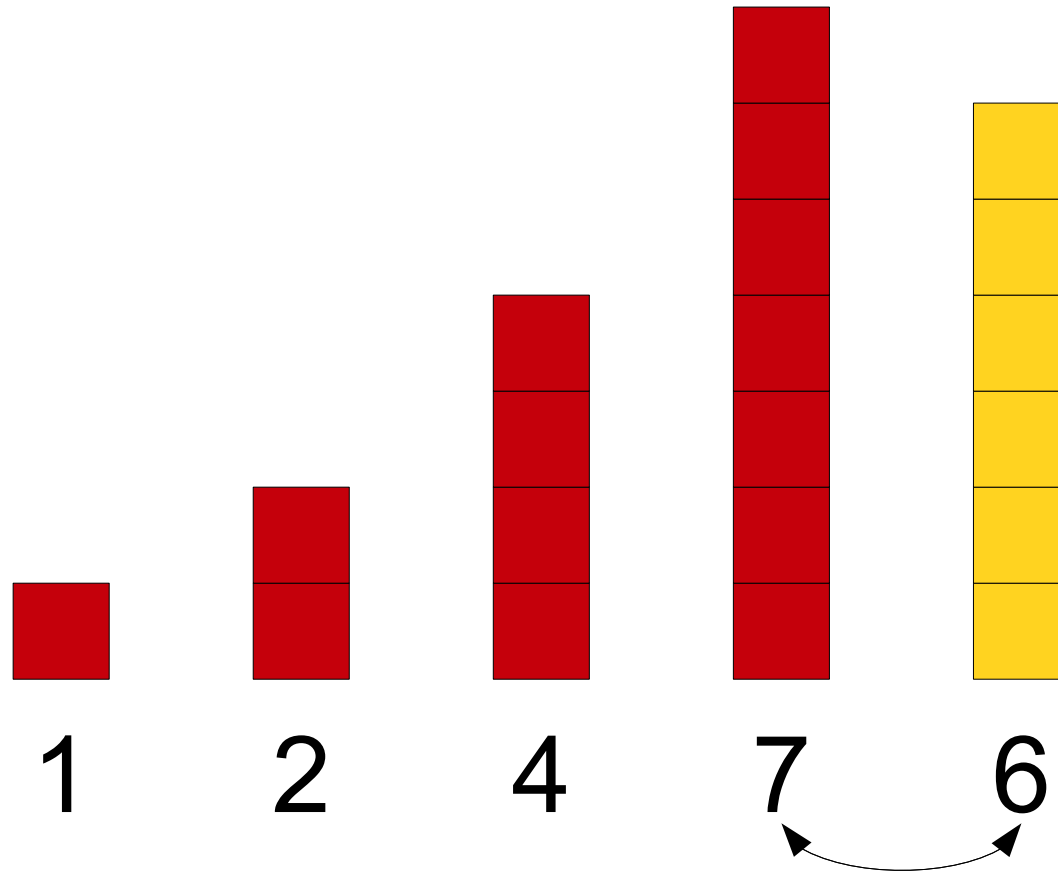
Insertion Sort



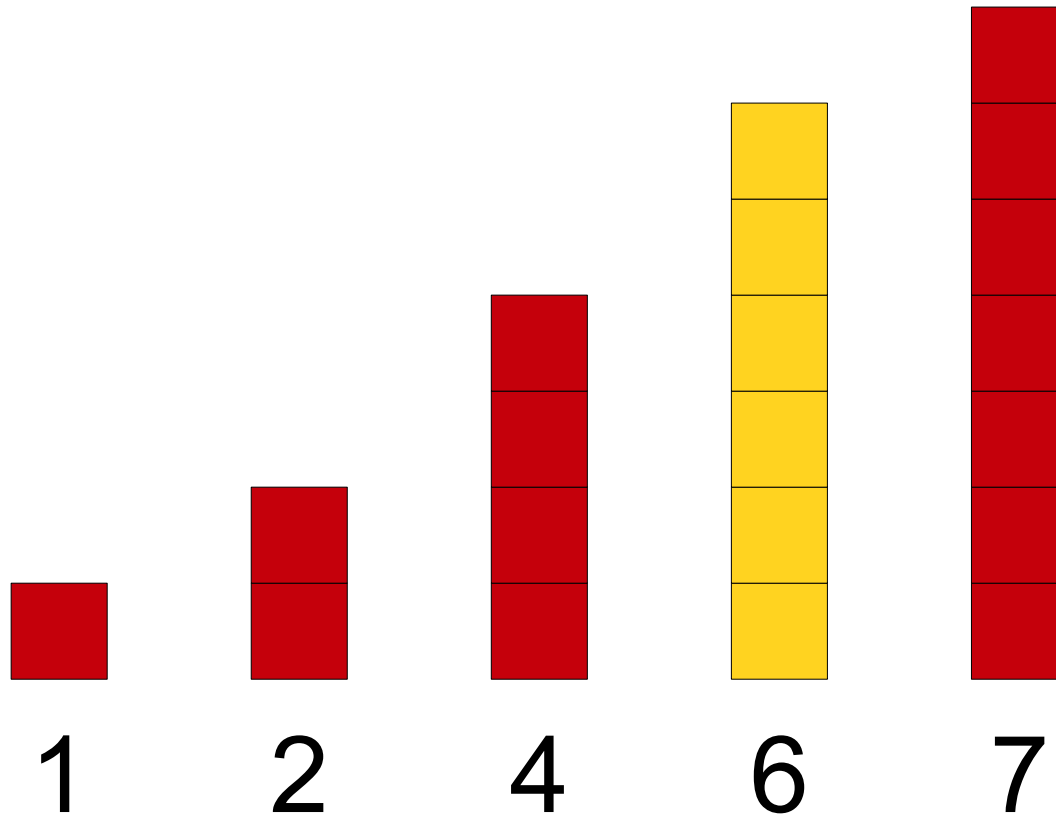
Insertion Sort



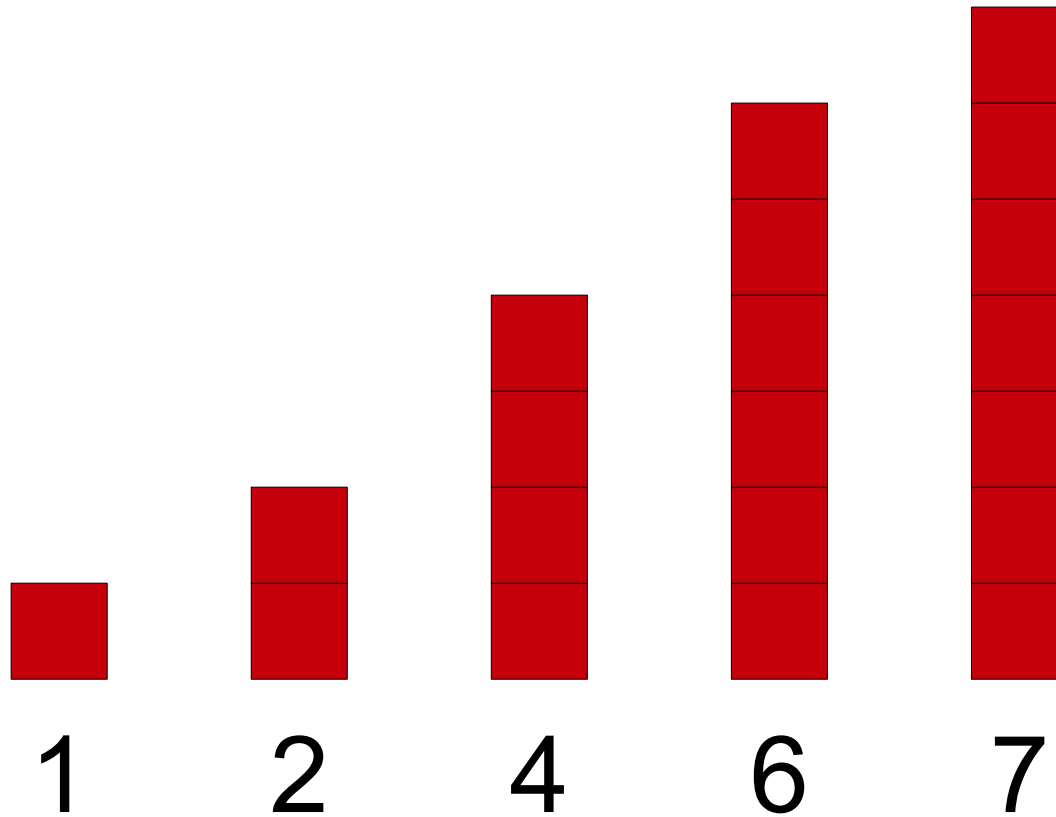
Insertion Sort



Insertion Sort



Insertion Sort



Selection Sort vs Insertion Sort

- Selection sort is $\Theta(n^2)$; it always takes quadratic time.
- Insertion is $O(n^2)$ in the worst case, but $O(n)$ in the best case.
- Consequently, insertion sort is usually faster than selection sort.
- Selection sort does more work at the beginning.
- Insertion sort does more work at the end.

Selection Sort vs Insertion Sort

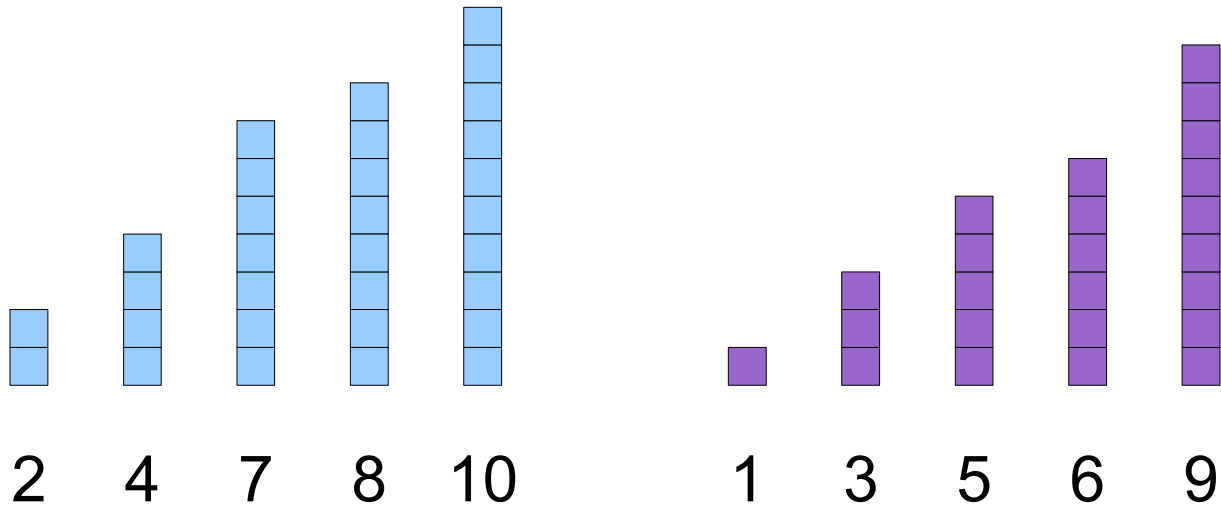
	Size	Selection Sort	Insertion Sort
	10000	0.304	0.160
	20000	1.218	0.630
	30000	2.790	1.427
	40000	4.646	2.520
	50000	7.395	4.181
	60000	10.584	5.635
	70000	14.149	8.143
	80000	18.674	10.333
	90000	23.165	12.832

A Note on $O(n^2)$

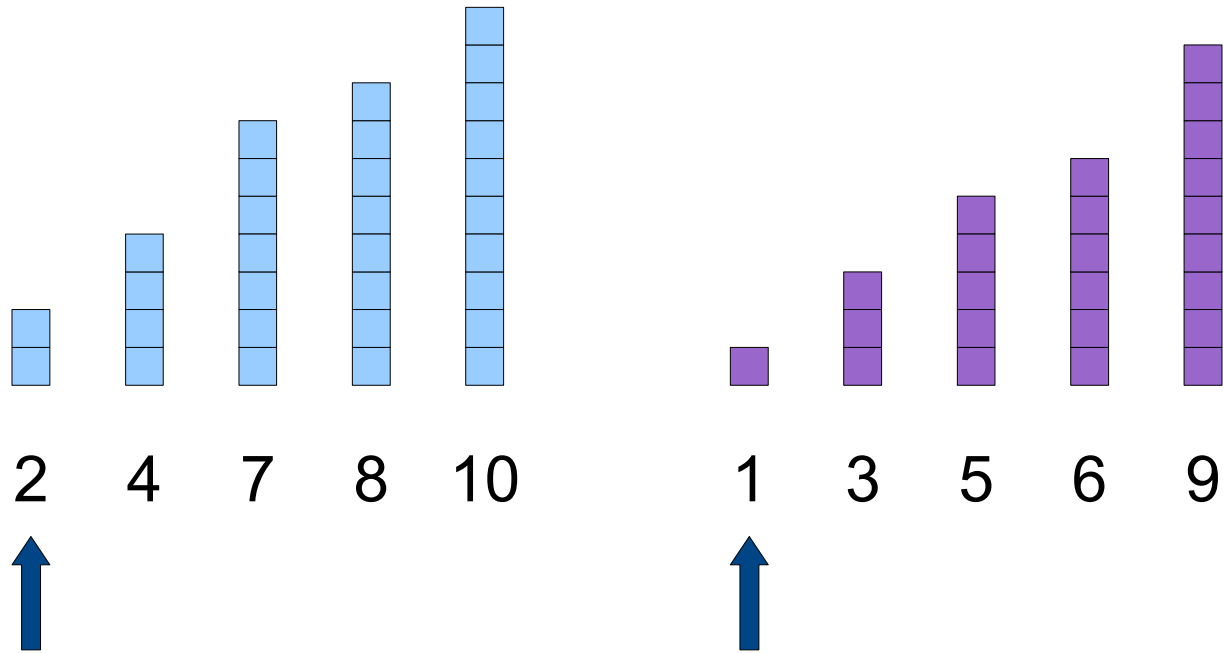
- If an algorithm is $O(n^2)$, what happens to the runtime if we double the input size?
 - Should go up by a factor of four: $(2n)^2 = 4n^2$
- If an algorithm is $O(n^2)$, what happens to the runtime if we *halve* the input size?
 - Should go *down* by a factor of four: $(\frac{1}{2}n)^2 = \frac{1}{4}n^2$
- It takes less work to sort the two halves of the array independently than it does to sort the entire array.
- Can we combine the results together?

The Key Insight: **Merge**

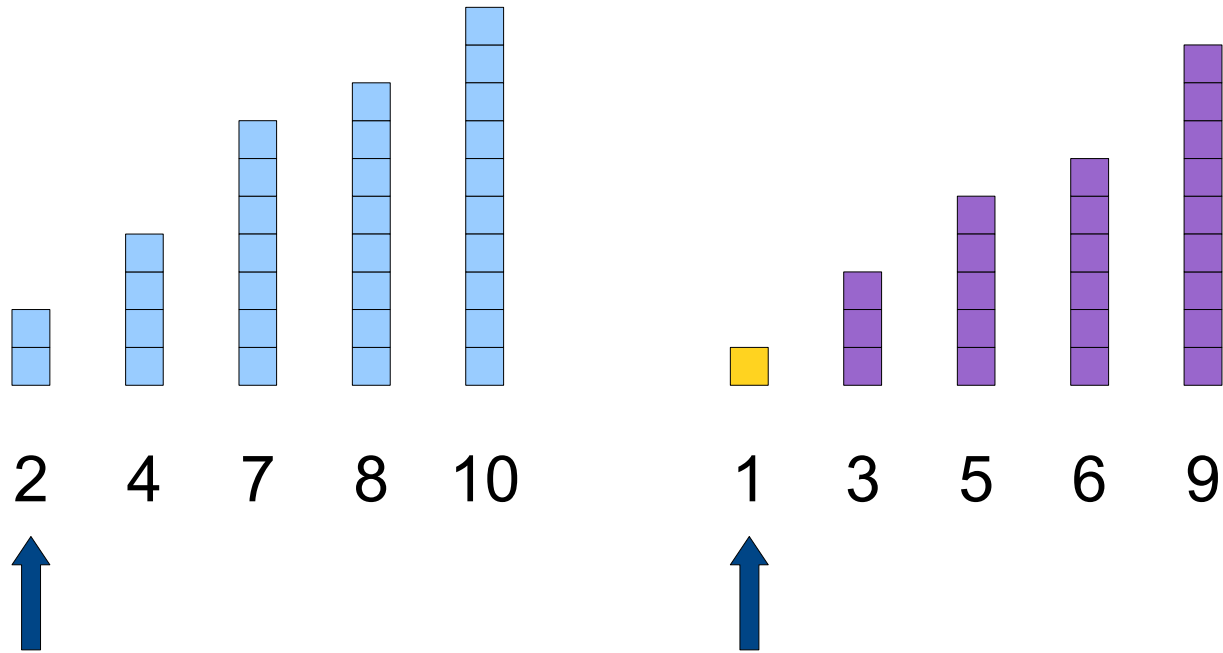
The Key Insight: Merge



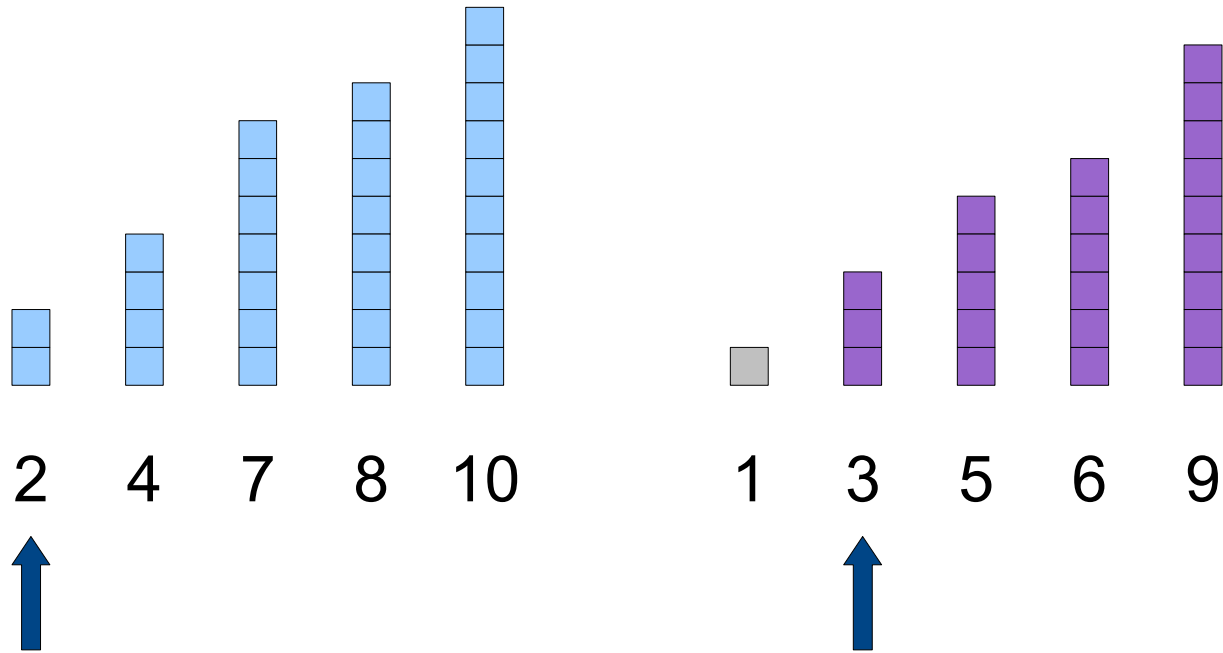
The Key Insight: Merge



The Key Insight: Merge

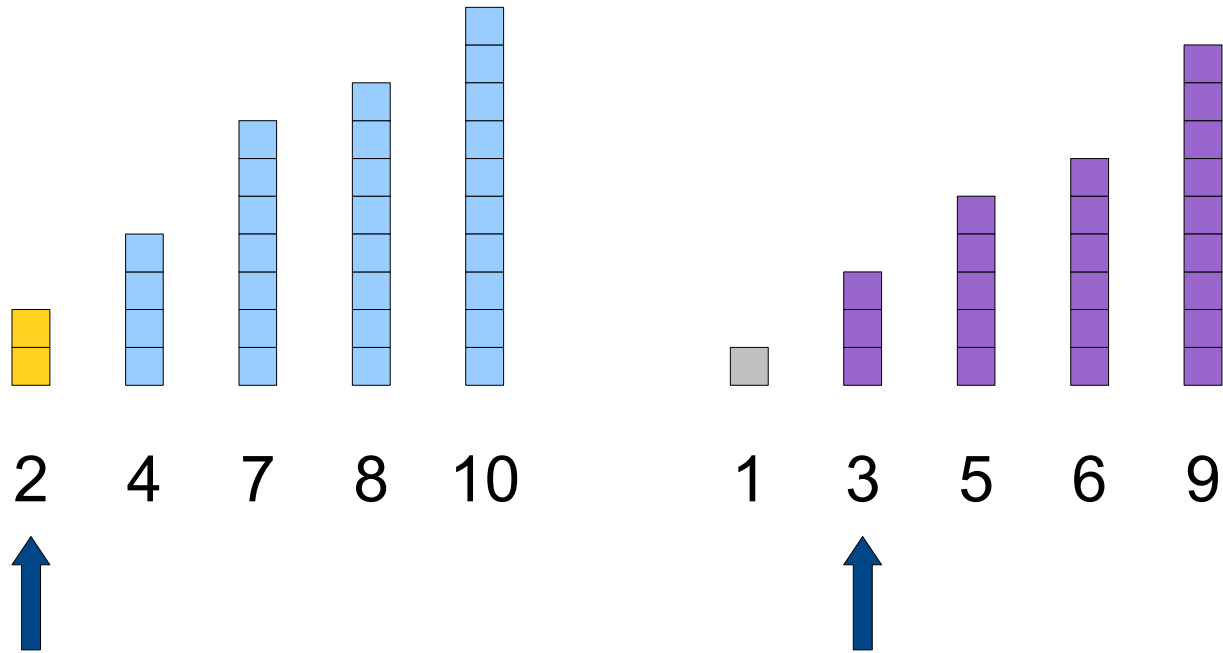


The Key Insight: Merge



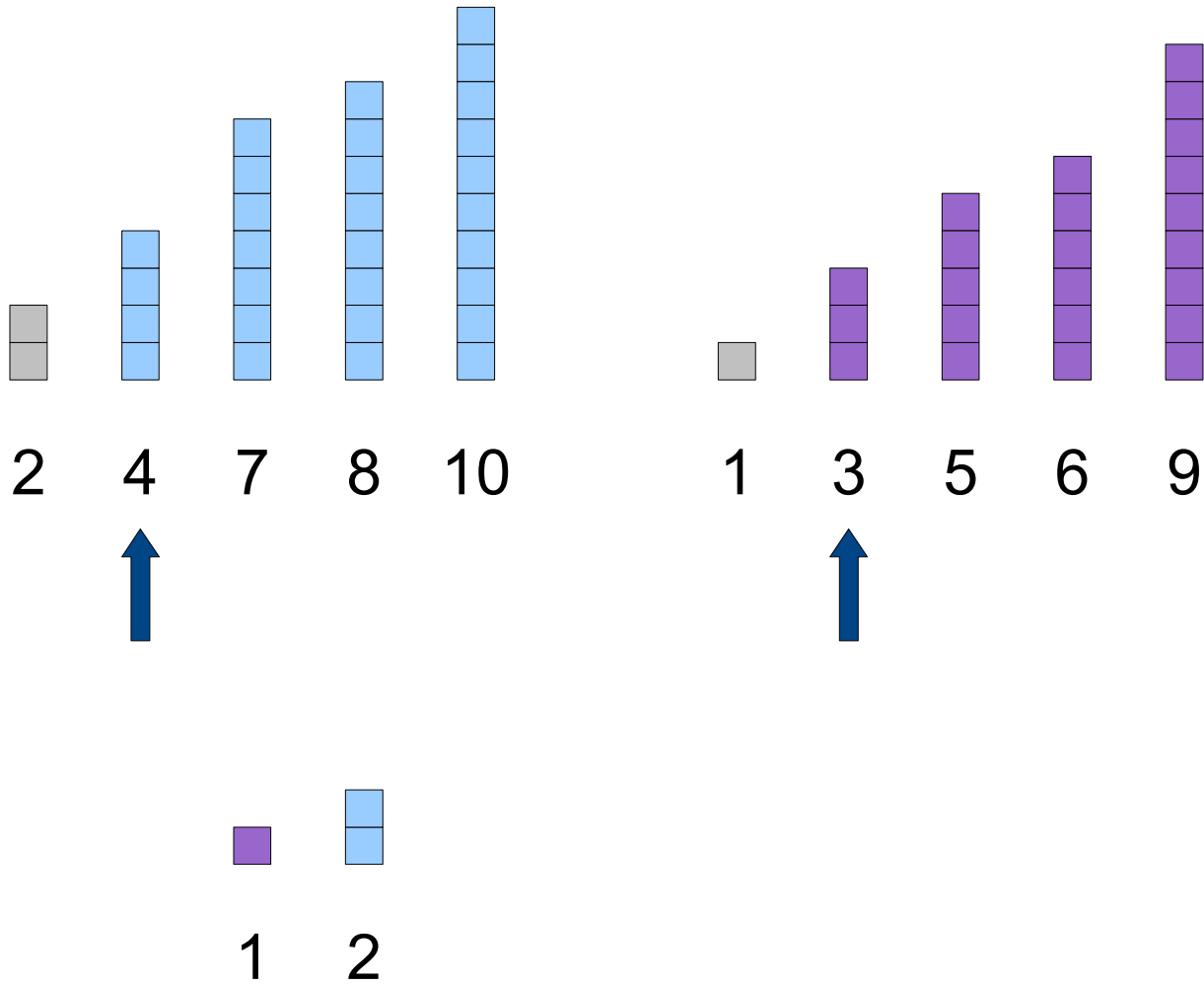
1

The Key Insight: Merge

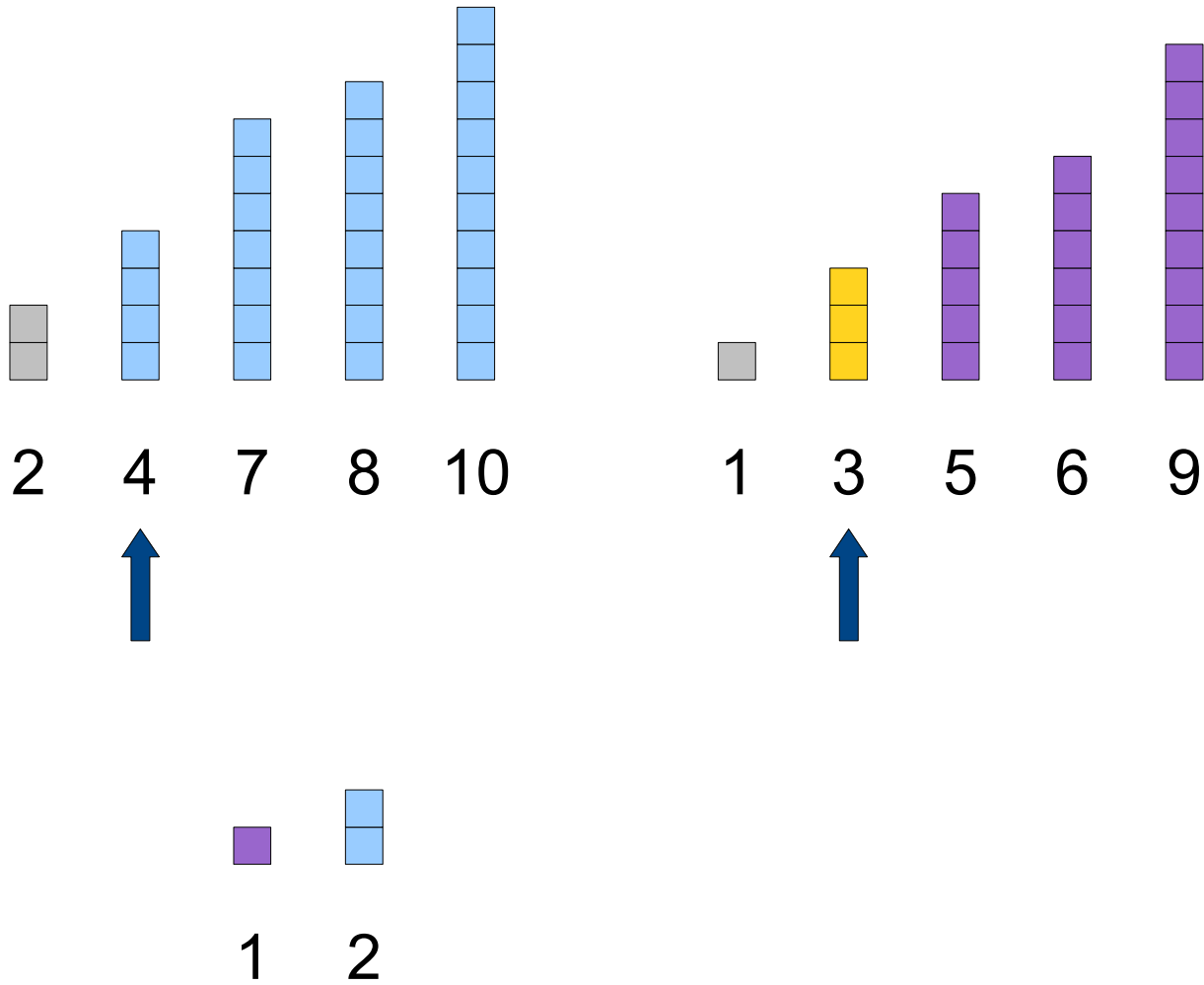


1

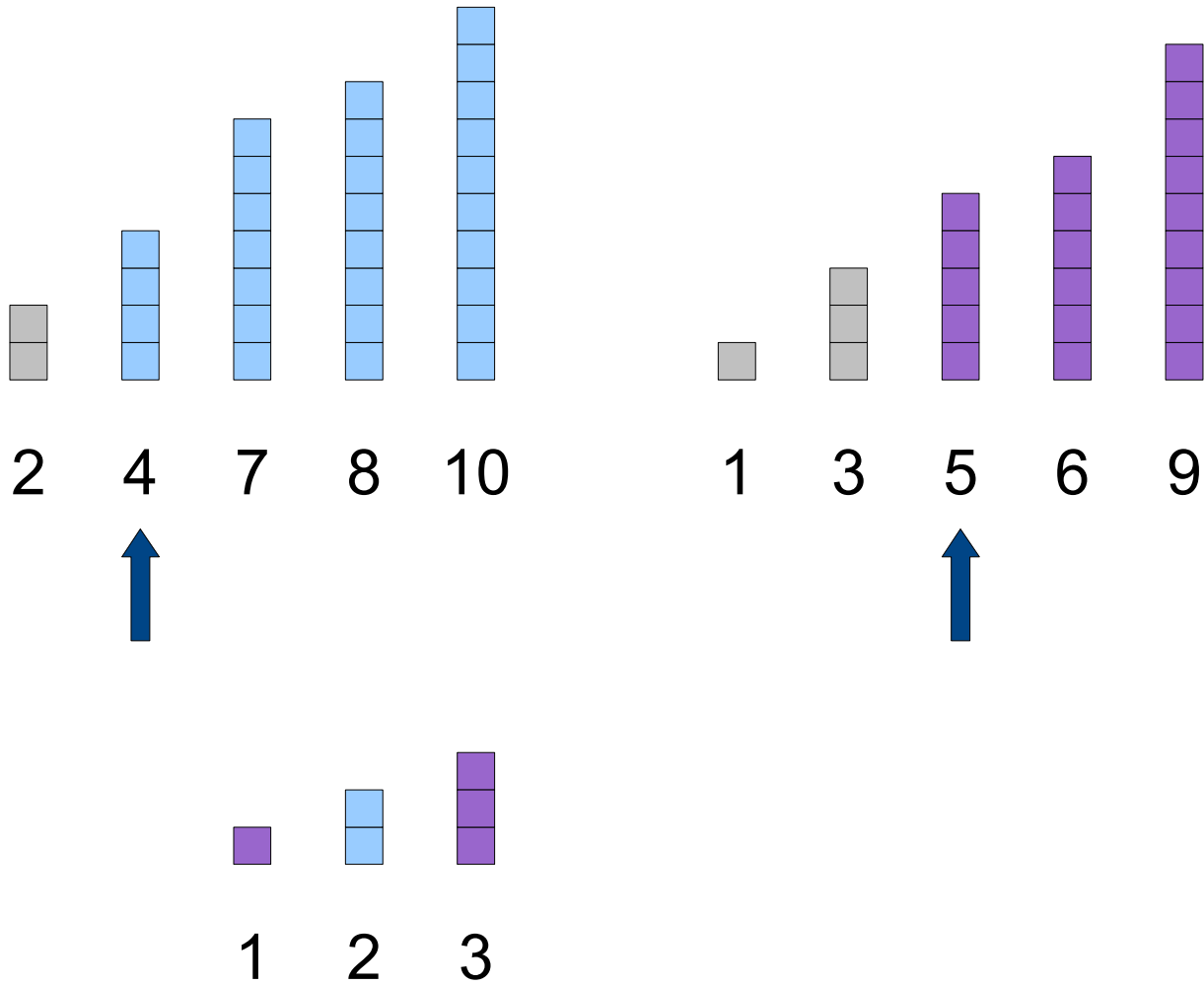
The Key Insight: Merge



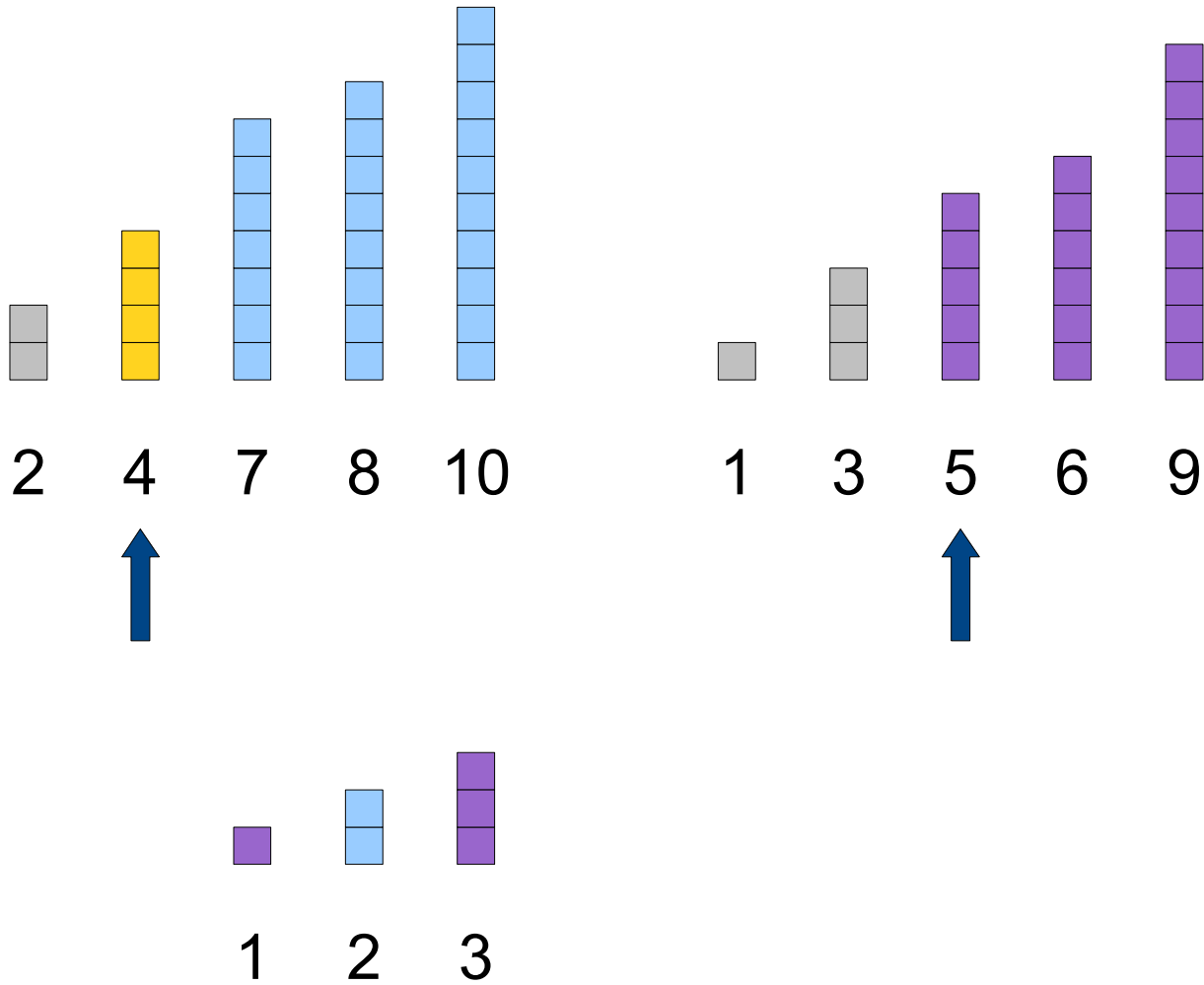
The Key Insight: Merge



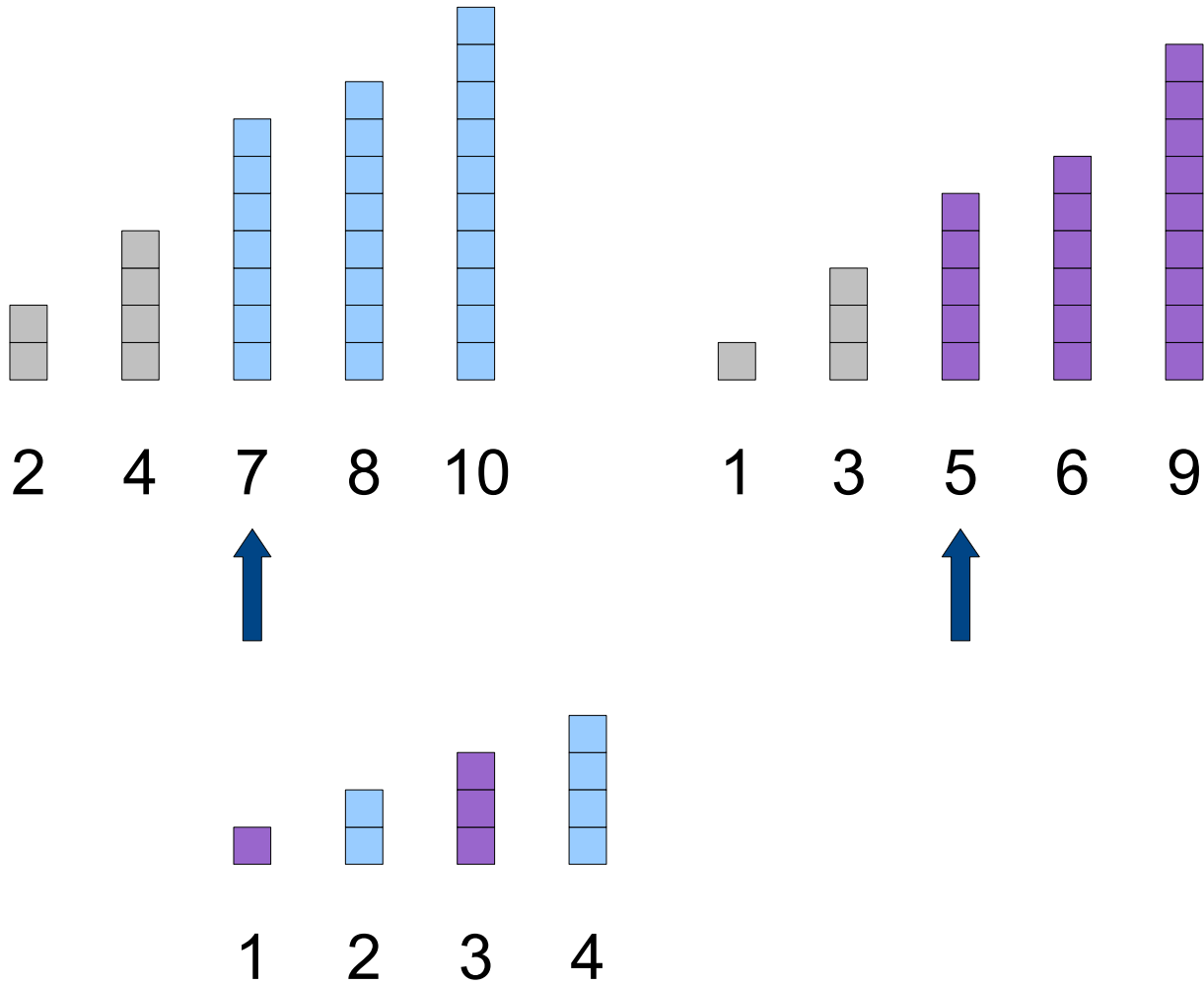
The Key Insight: Merge



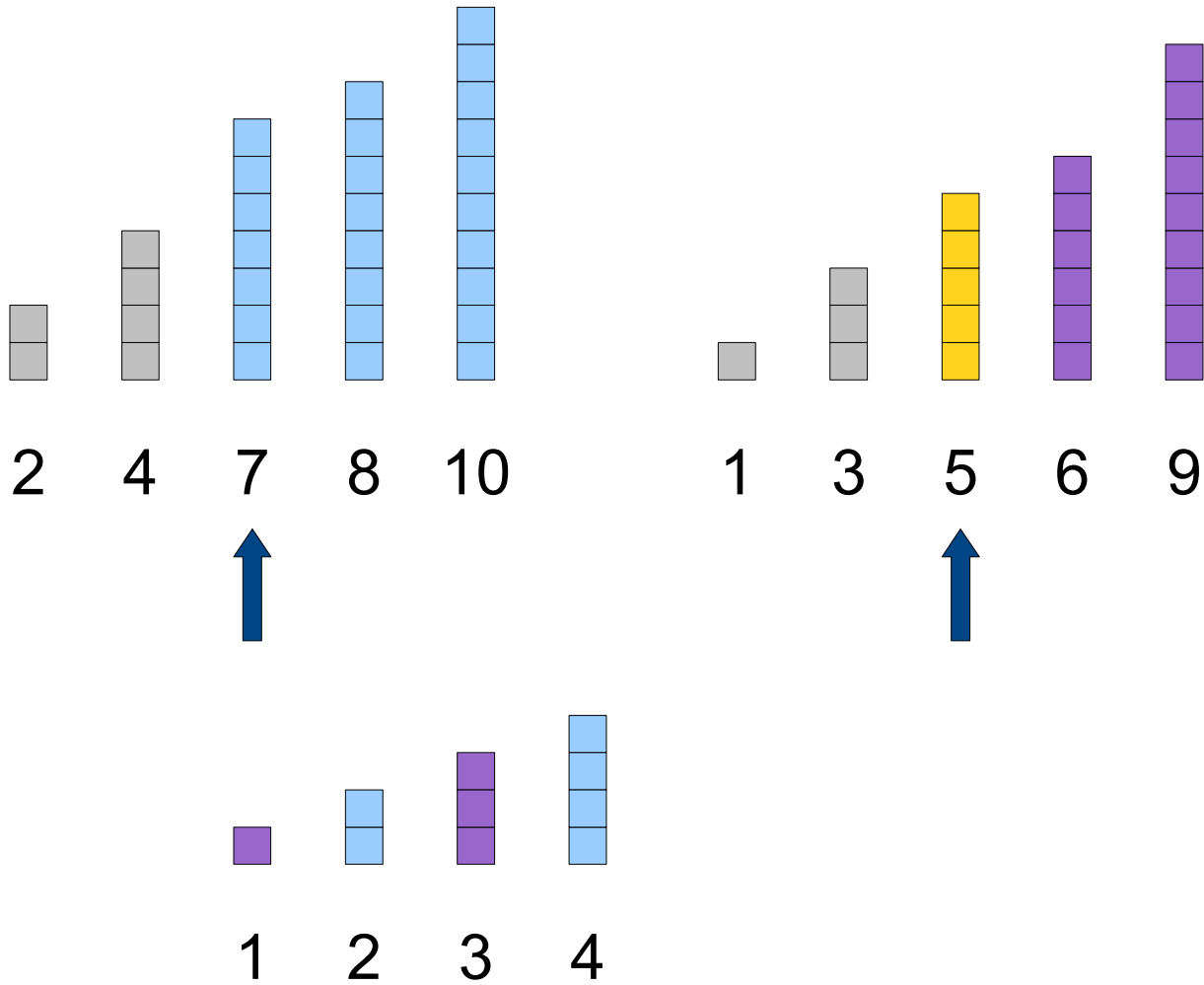
The Key Insight: Merge



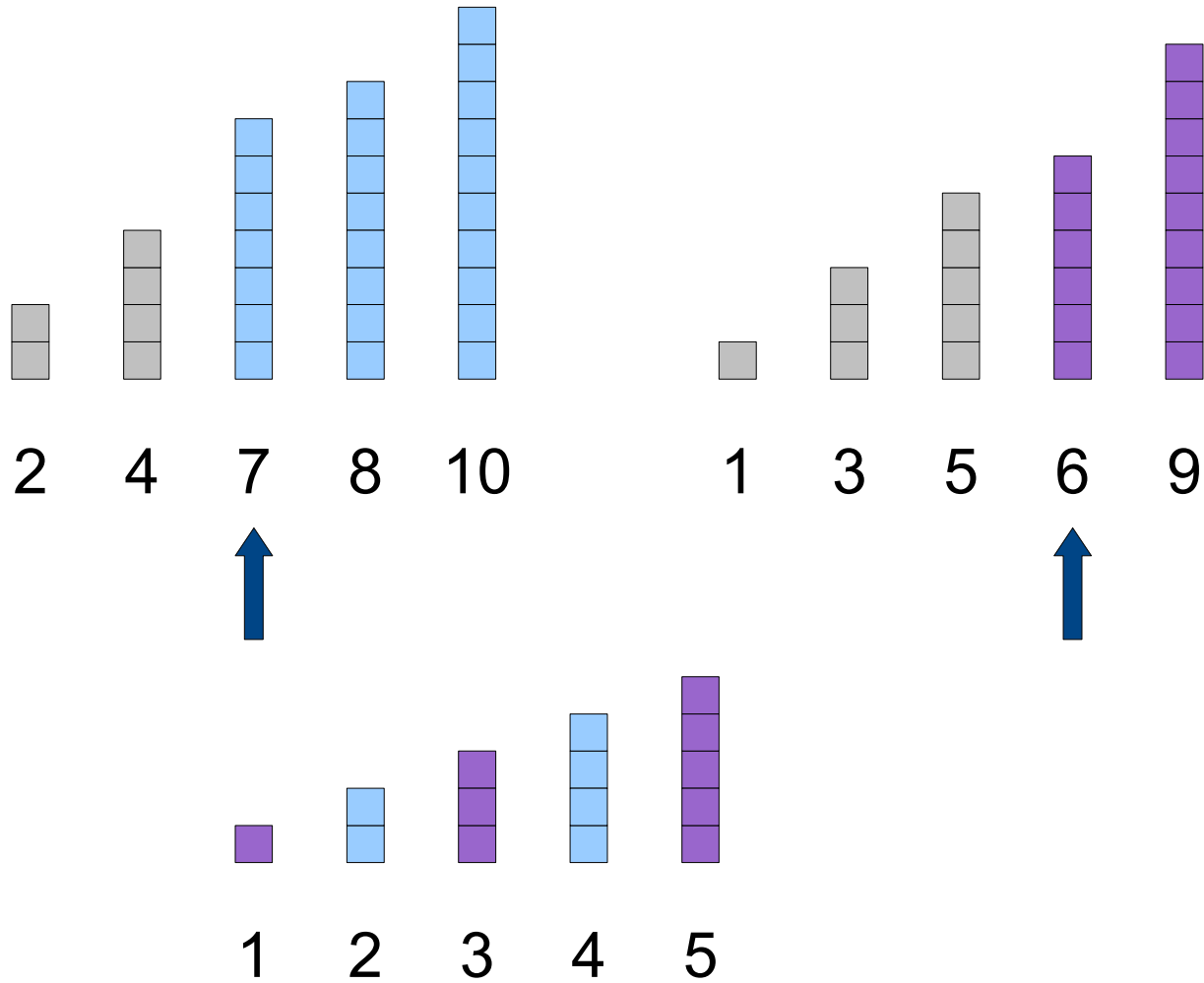
The Key Insight: Merge



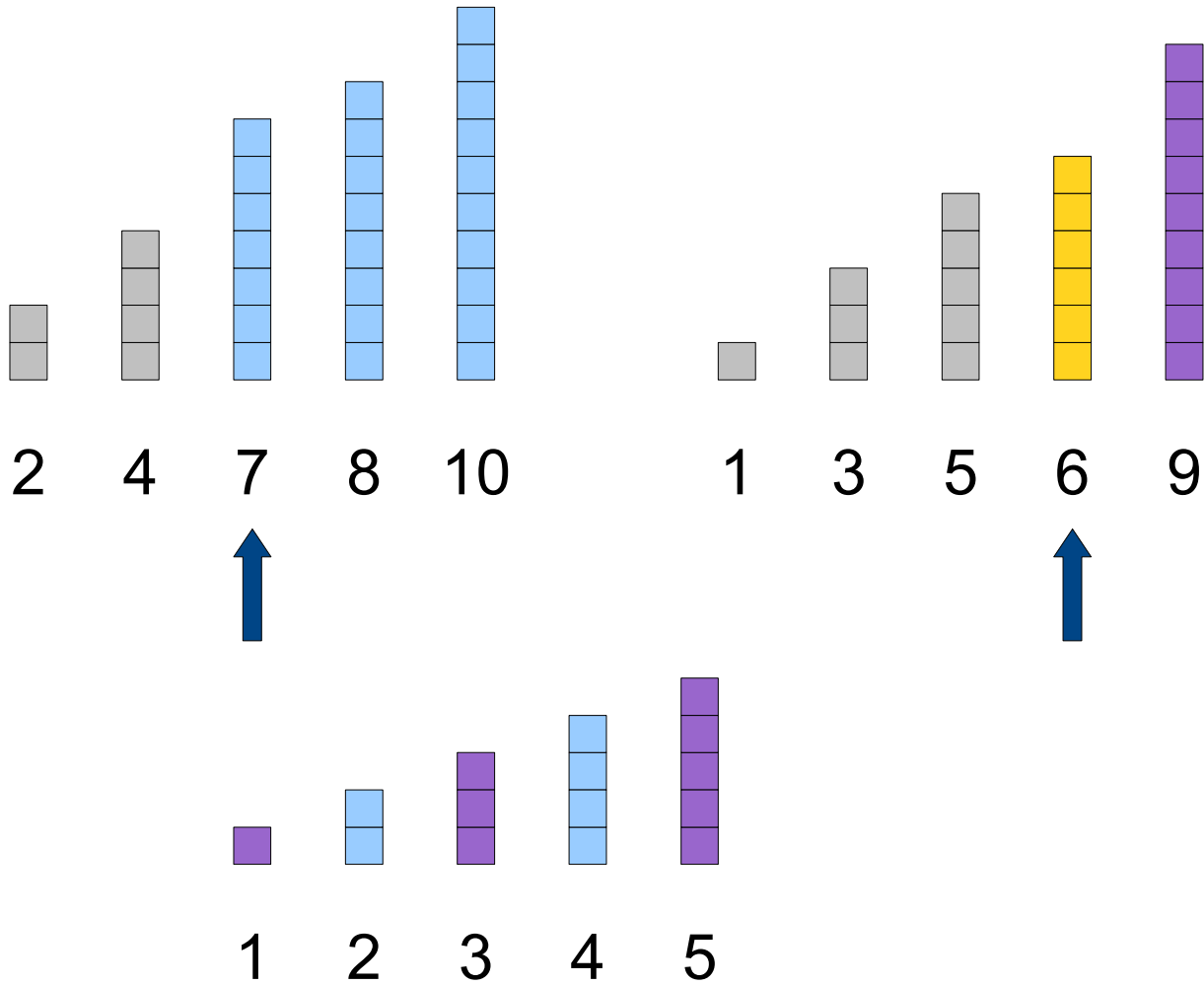
The Key Insight: Merge



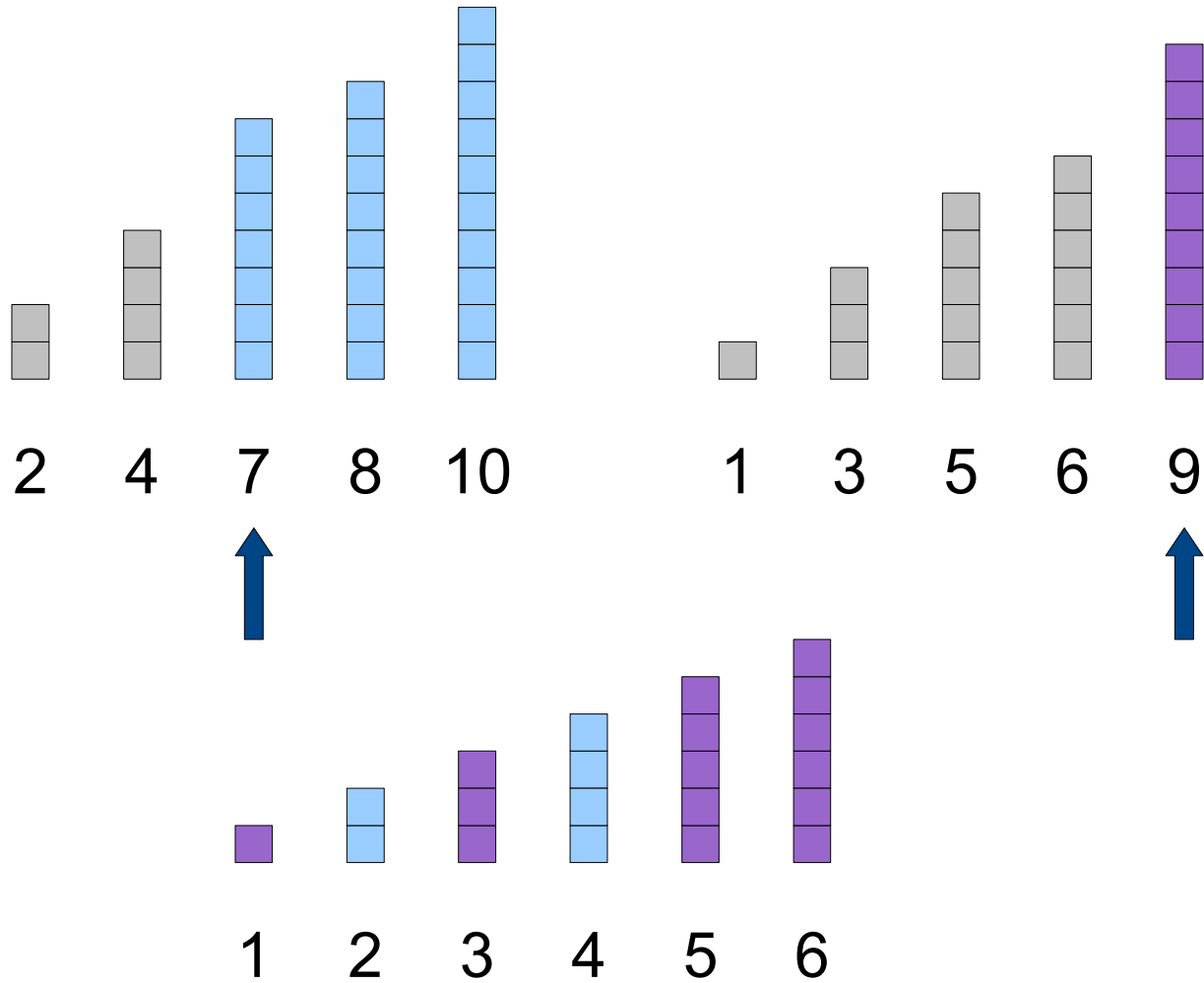
The Key Insight: Merge



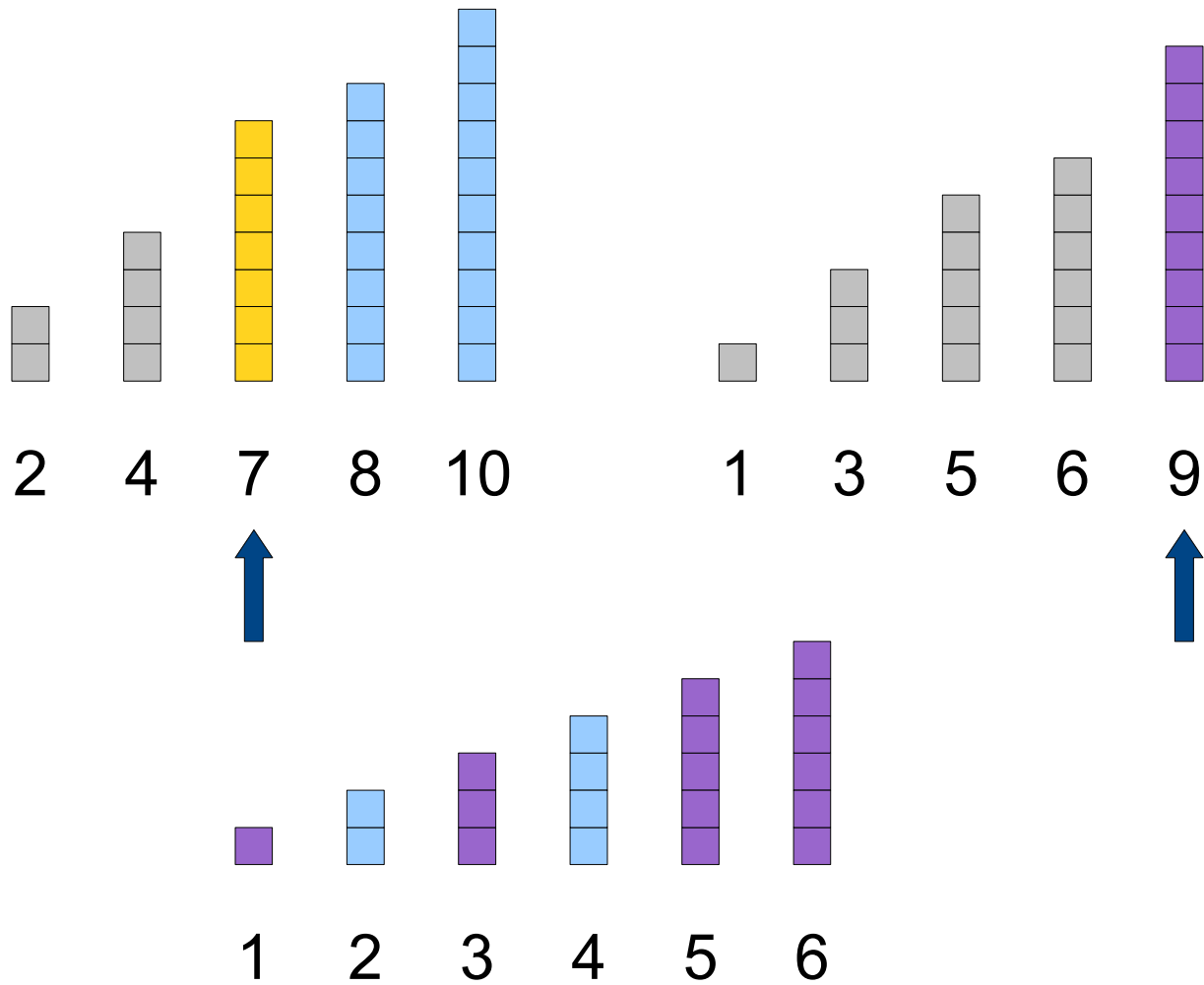
The Key Insight: Merge



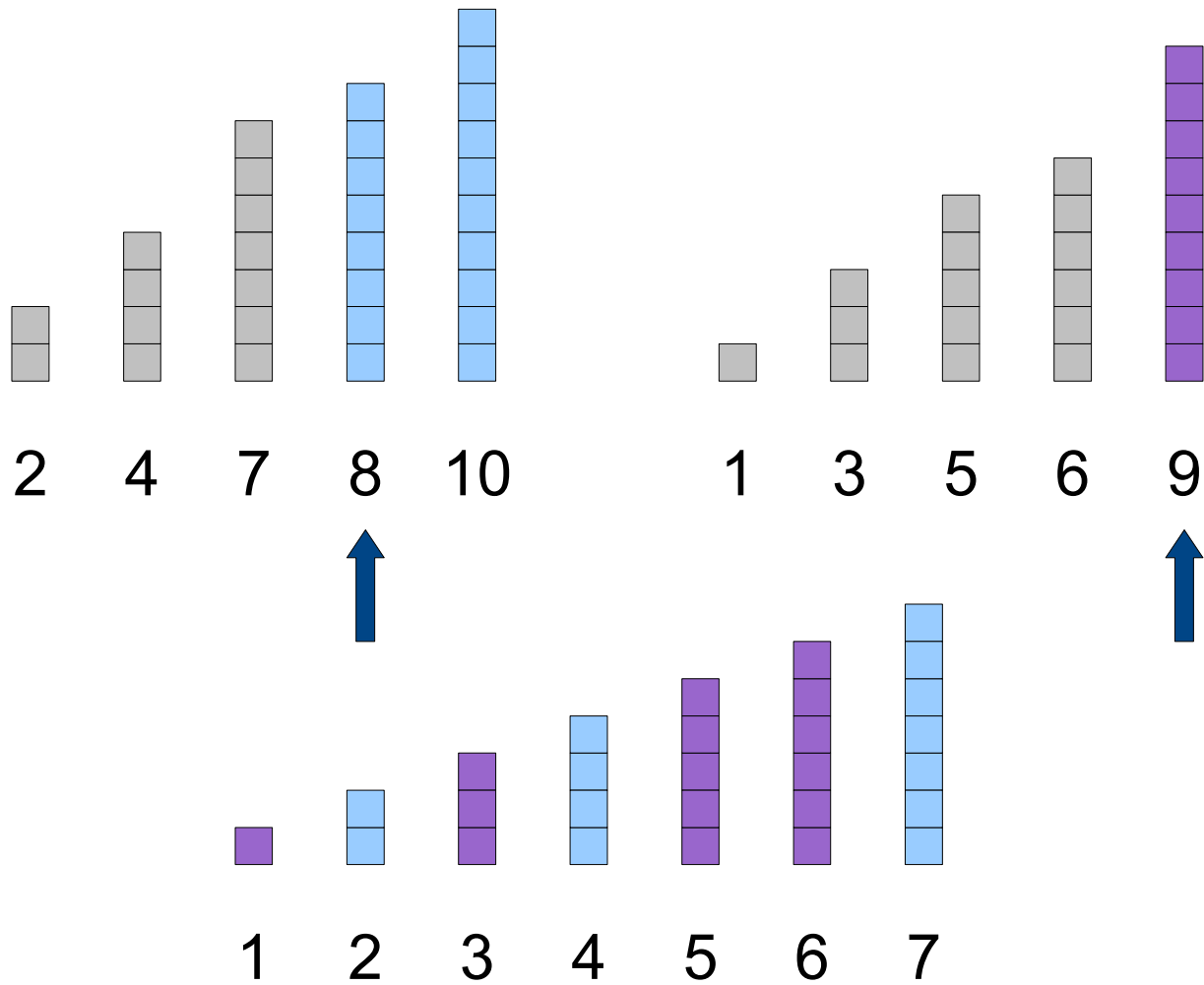
The Key Insight: Merge



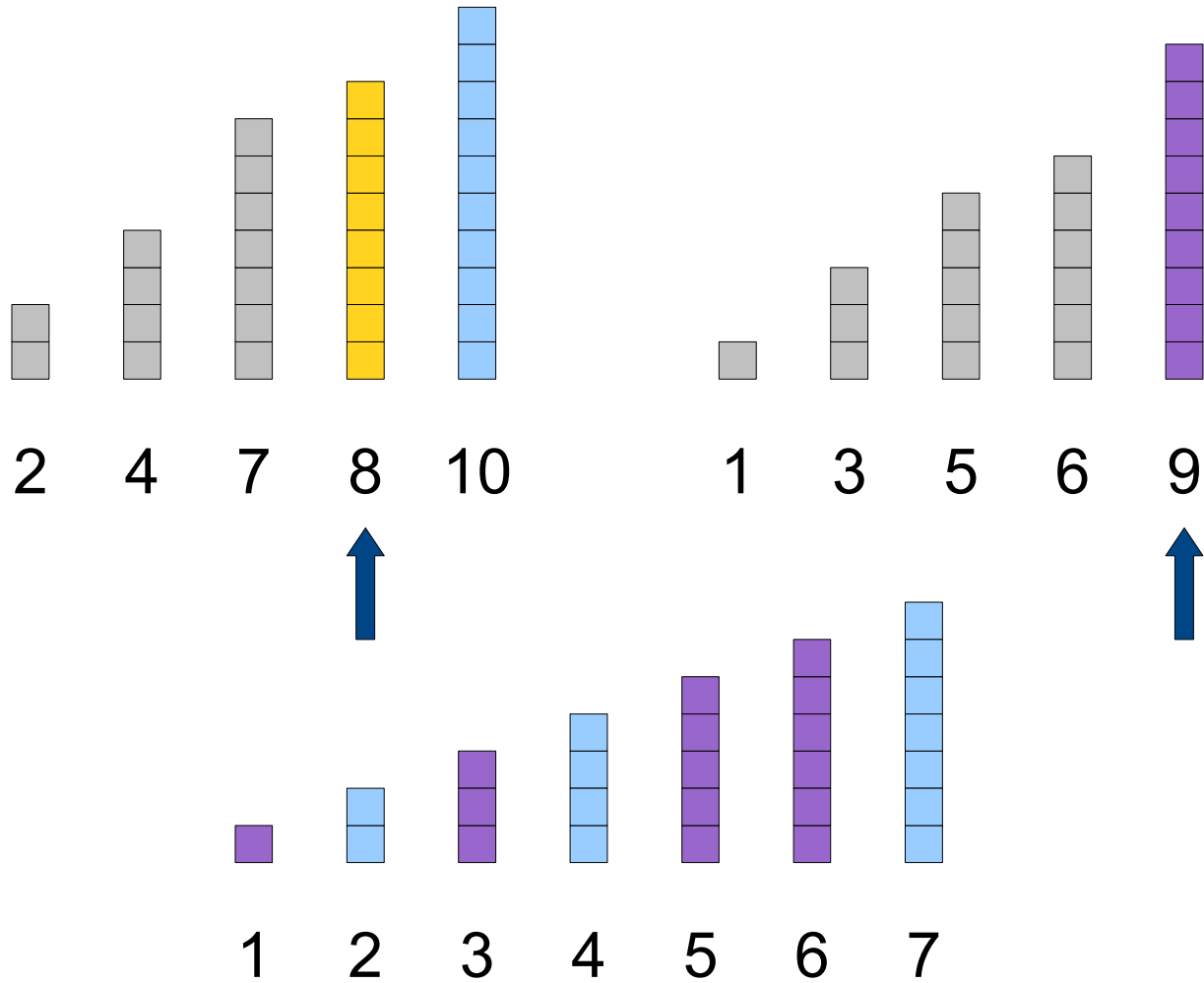
The Key Insight: Merge



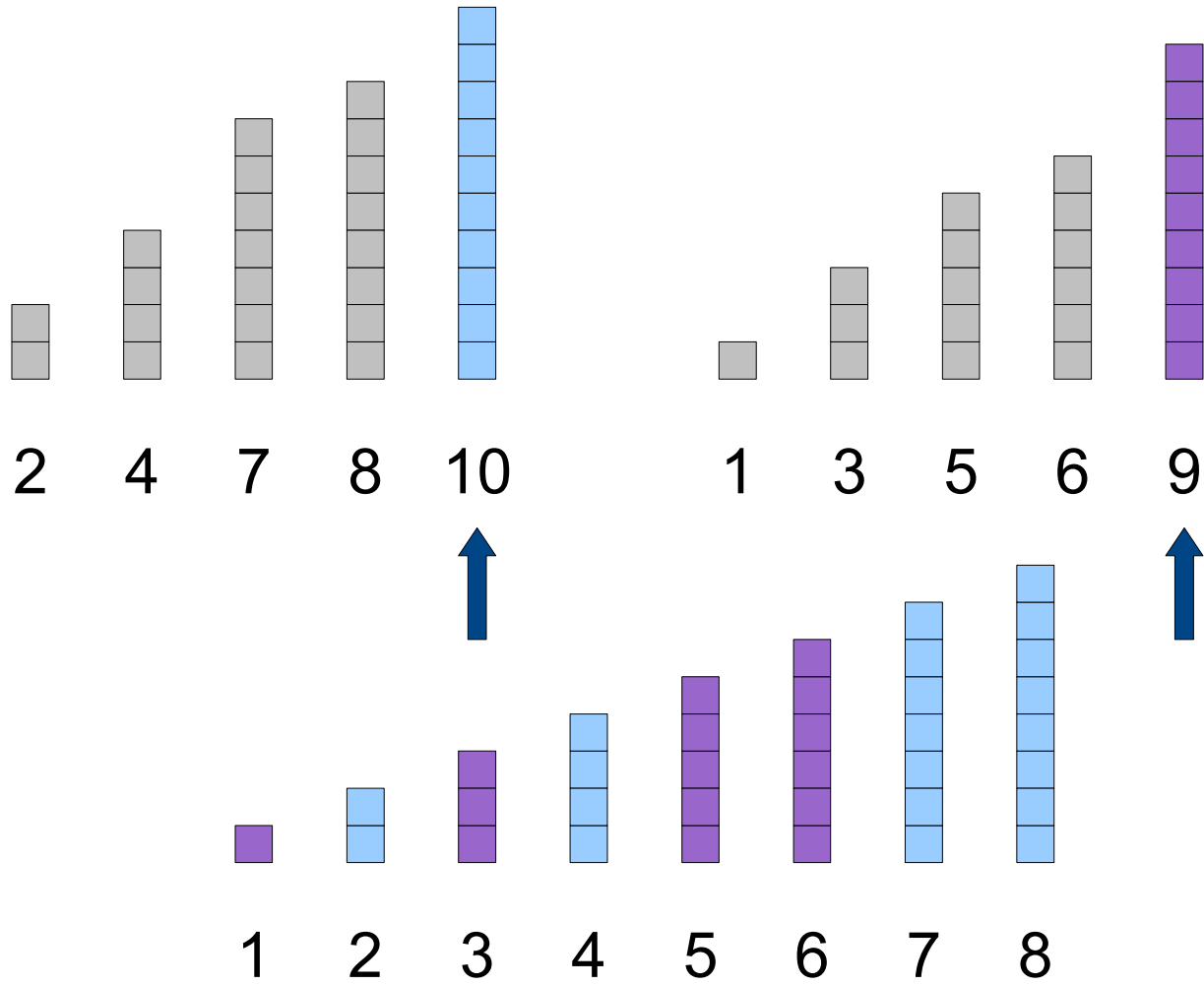
The Key Insight: Merge



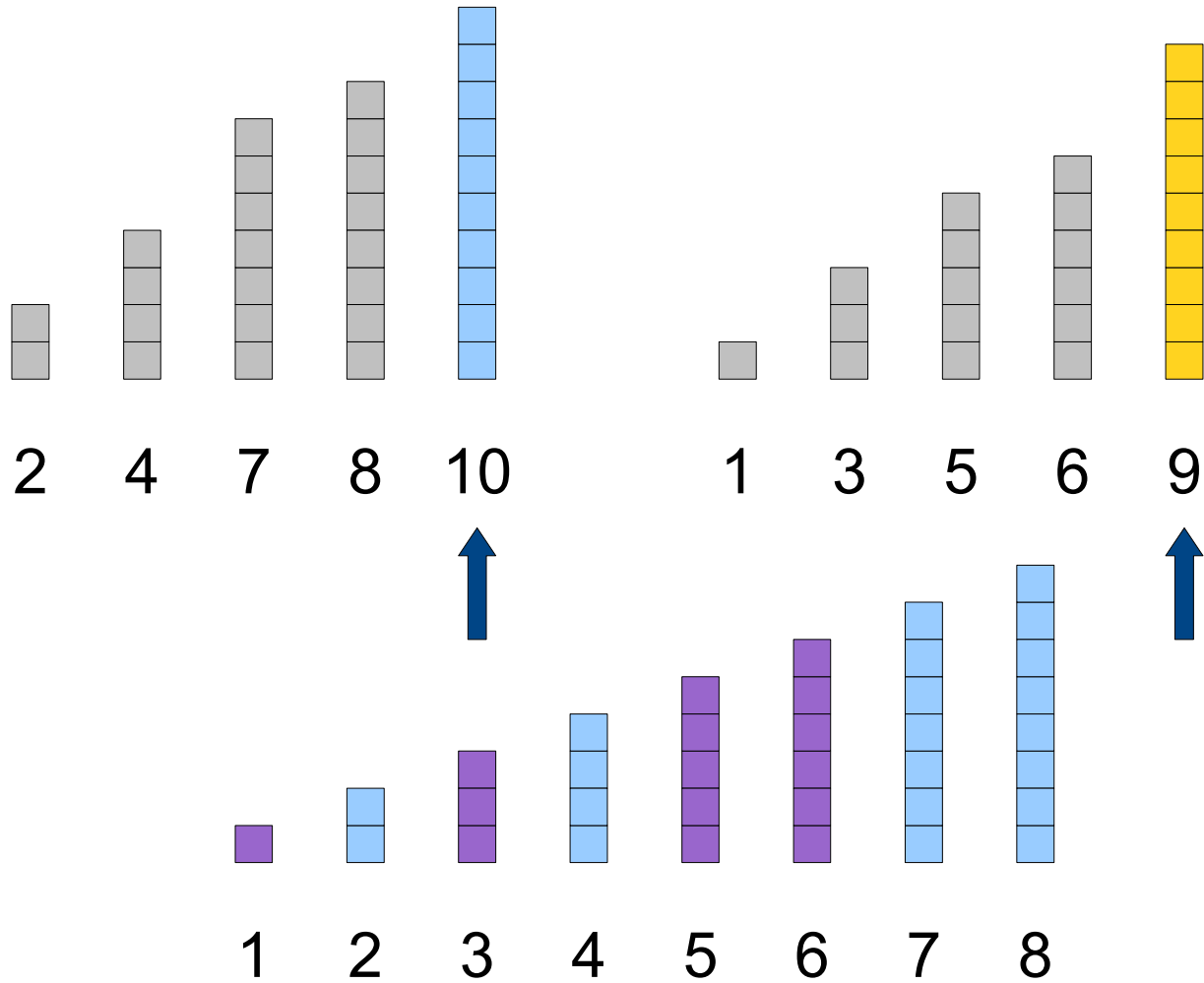
The Key Insight: Merge



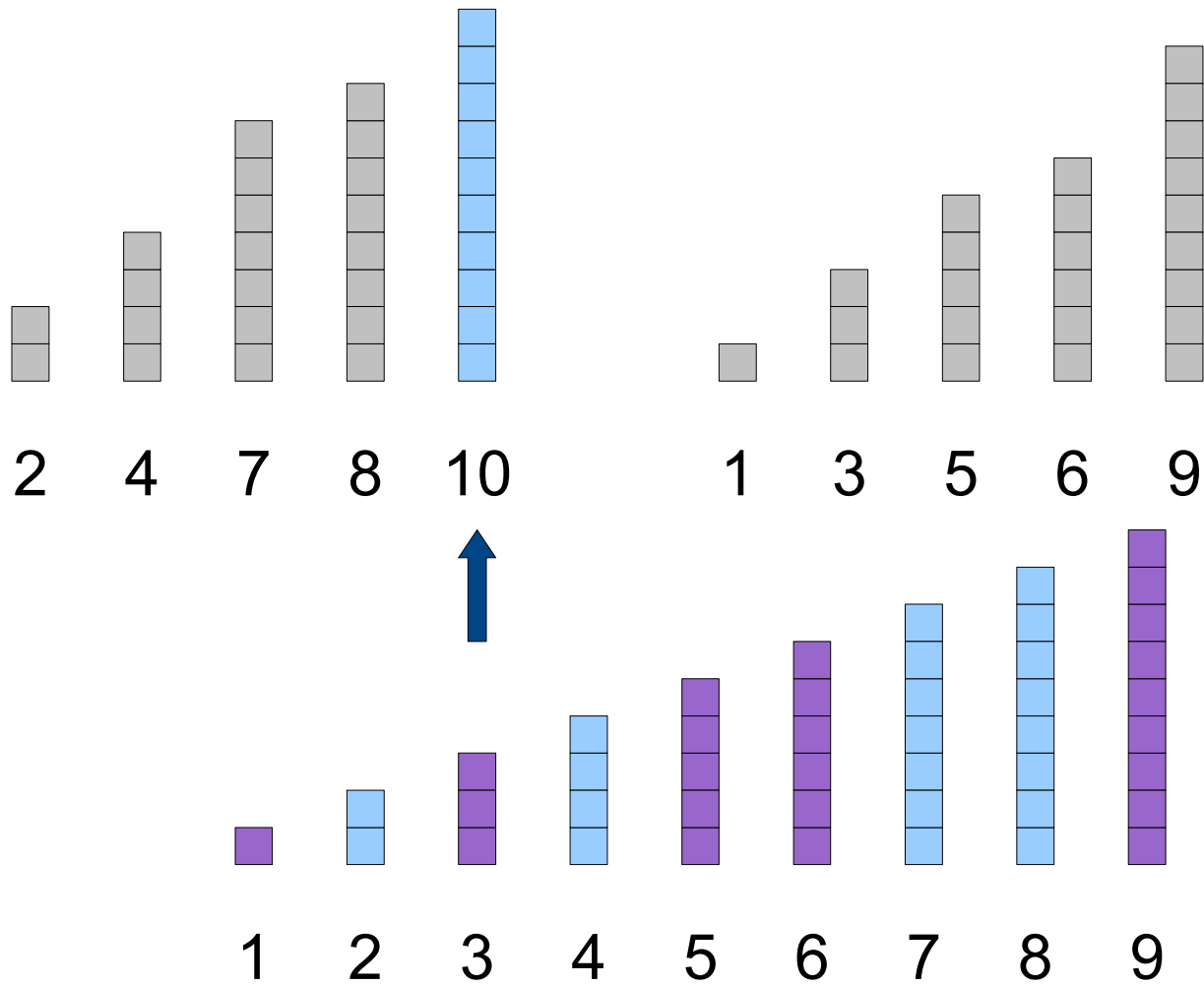
The Key Insight: Merge



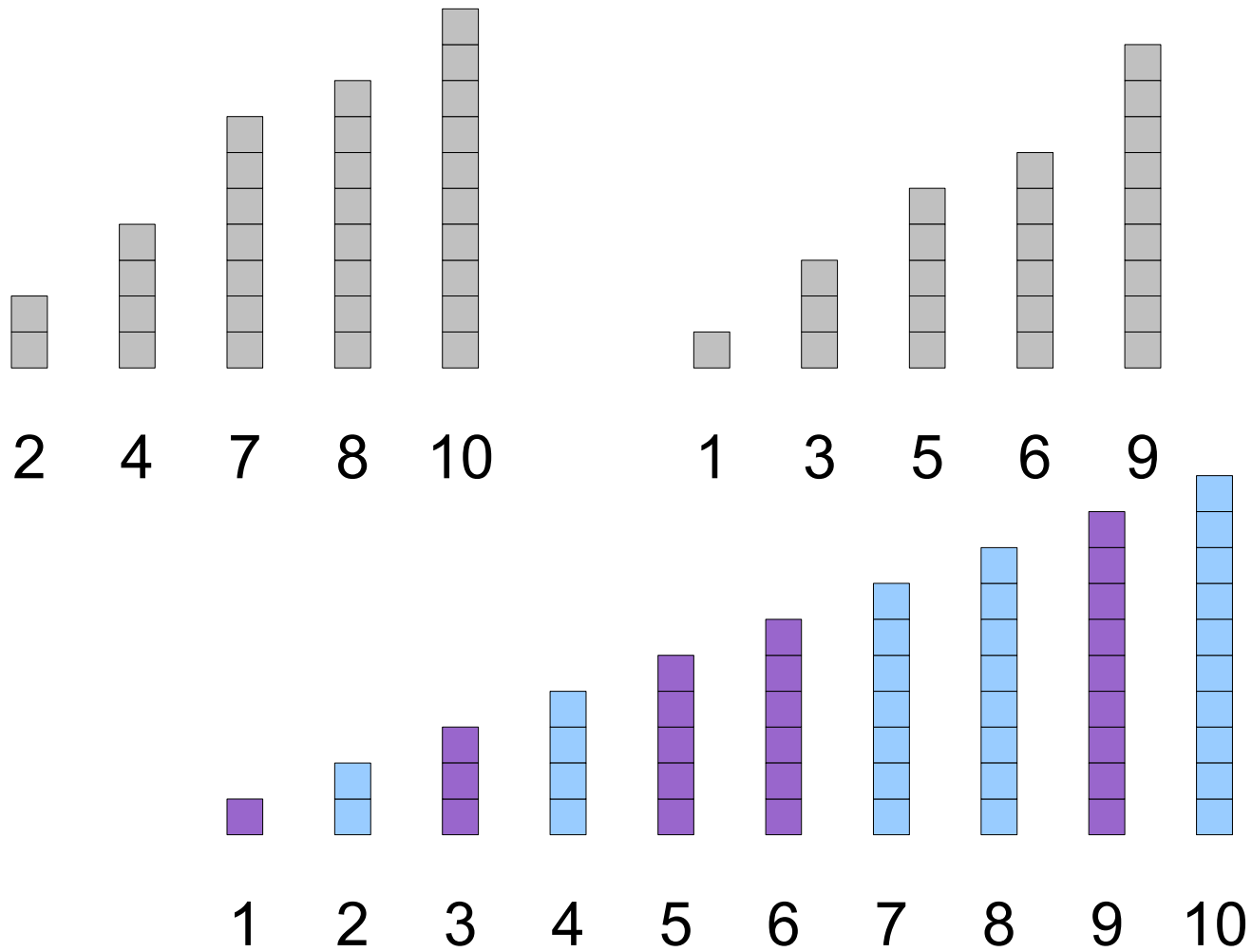
The Key Insight: Merge



The Key Insight: Merge



The Key Insight: Merge



Code for Merge

Code for Merge

```
void Merge(Vector<int>& one, Vector<int>& two, Vector<int>& result)
{
    result.clear();

    int oneRead = 0, twoRead = 0;
    while (oneRead != one.size() && twoRead != two.size()) {
        if (one[oneRead] < two[twoRead]) {
            result.add(one[oneRead]);
            oneRead++;
        } else {
            result.add(two[twoRead]);
            twoRead++;
        }
    }

    for(; oneRead != one.size(); ++oneRead)
        result.add(one[oneRead]);
    for(; twoRead != two.size(); ++twoRead)
        result.add(two[twoRead]);
}
```

Code for Merge

```
void Merge(Vector<int>& one, Vector<int>& two, Vector<int>& result)
{
    result.clear();

    int oneRead = 0, twoRead = 0;
    while (oneRead != one.size() && twoRead != two.size()) {
        if (one[oneRead] < two[twoRead]) {
            result.add(one[oneRead]);
            oneRead++;
        } else {
            result.add(two[twoRead]);
            twoRead++;
        }
    }

    for(; oneRead != one.size(); ++oneRead)
        result.add(one[oneRead]);
    for(; twoRead != two.size(); ++twoRead)
        result.add(two[twoRead]);
}
```

Code for Merge

```
void Merge(Vector<int>& one, Vector<int>& two, Vector<int>& result)
{
    result.clear();

    int oneRead = 0, twoRead = 0;
    while (oneRead != one.size() && twoRead != two.size()) {
        if (one[oneRead] < two[twoRead]) {
            result.add(one[oneRead]);
            oneRead++;
        } else {
            result.add(two[twoRead]);
            twoRead++;
        }
    }

    for(; oneRead != one.size(); ++oneRead)
        result.add(one[oneRead]);
    for(; twoRead != two.size(); ++twoRead)
        result.add(two[twoRead]);
}
```

Code for Merge

```
void Merge(Vector<int>& one, Vector<int>& two, Vector<int>& result)
{
    result.clear();

    int oneRead = 0, twoRead = 0;
    while (oneRead != one.size() && twoRead != two.size()) {
        if (one[oneRead] < two[twoRead]) {
            result.add(one[oneRead]);
            oneRead++;
        } else {
            result.add(two[twoRead]);
            twoRead++;
        }
    }

    for(; oneRead != one.size(); ++oneRead)
        result.add(one[oneRead]);
    for(; twoRead != two.size(); ++twoRead)
        result.add(two[twoRead]);
}
```

Code for Merge

```
void Merge(Vector<int>& one, Vector<int>& two, Vector<int>& result)
{
    result.clear();

    int oneRead = 0, twoRead = 0;
    while (oneRead != one.size() && twoRead != two.size()) {
        if (one[oneRead] < two[twoRead]) {
            result.add(one[oneRead]);
            oneRead++;
        } else {
            result.add(two[twoRead]);
            twoRead++;
        }
    }

    for(; oneRead != one.size(); ++oneRead)
        result.add(one[oneRead]);
    for(; twoRead != two.size(); ++twoRead)
        result.add(two[twoRead]);
}
```

Code for Merge

```
void Merge(Vector<int>& one, Vector<int>& two, Vector<int>& result)
{
    result.clear();

    int oneRead = 0, twoRead = 0;
    while (oneRead != one.size() && twoRead != two.size()) {
        if (one[oneRead] < two[twoRead]) {
            result.add(one[oneRead]);
            oneRead++;
        } else {
            result.add(two[twoRead]);
            twoRead++;
        }
    }

    for(; oneRead != one.size(); ++oneRead)
        result.add(one[oneRead]);
    for(; twoRead != two.size(); ++twoRead)
        result.add(two[twoRead]);
}
```

Code for Merge

```
void Merge(Vector<int>& one, Vector<int>& two, Vector<int>& result)
{
    result.clear();

    int oneRead = 0, twoRead = 0;
    while (oneRead != one.size() && twoRead != two.size()) {
        if (one[oneRead] < two[twoRead]) {
            result.add(one[oneRead]);
            oneRead++;
        } else {
            result.add(two[twoRead]);
            twoRead++;
        }
    }

    for(; oneRead != one.size(); ++oneRead)
        result.add(one[oneRead]);
    for(; twoRead != two.size(); ++twoRead)
        result.add(two[twoRead]);
}
```

Code for Merge

```
void Merge(Vector<int>& one, Vector<int>& two, Vector<int>& result)
{
    result.clear();

    int oneRead = 0, twoRead = 0;
    while (oneRead != one.size() && twoRead != two.size()) {
        if (one[oneRead] < two[twoRead]) {
            result.add(one[oneRead]);
            oneRead++;
        } else {
            result.add(two[twoRead]);
            twoRead++;
        }
    }

    for(; oneRead != one.size(); ++oneRead)
        result.add(one[oneRead]);
    for(; twoRead != two.size(); ++twoRead)
        result.add(two[twoRead]);
}
```

An Initial Approach

- Split the input in half.
- Use insertion sort to sort each half.
- Use merge to combine them together.
- Runtime?
 - Still $O(n^2)$
 - Should be faster than regular insertion sort, though.

“Split Sort”

“Split Sort”

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

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

    InsertionSort(left);
    InsertionSort(right);

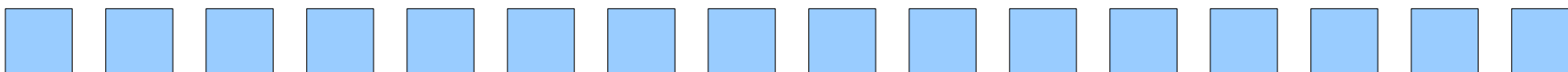
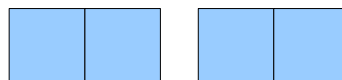
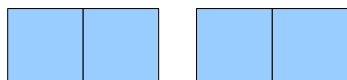
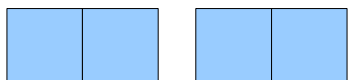
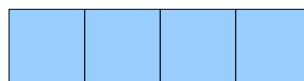
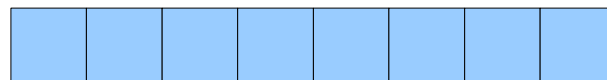
    Merge(left, right, v);
}
```

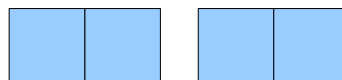
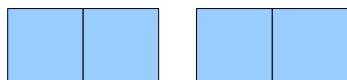
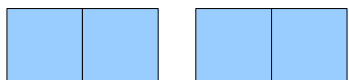
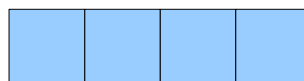
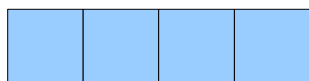
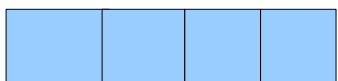
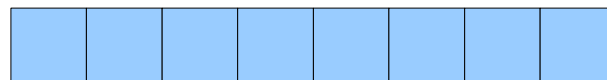
Performance Comparison

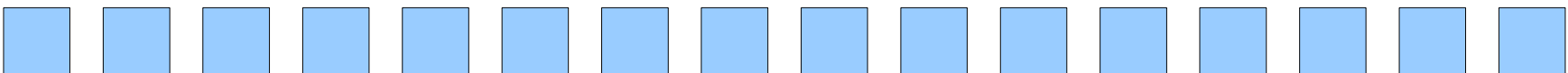
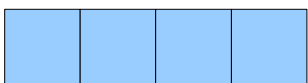
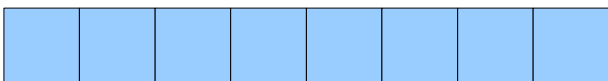
Size	Selection Sort	Insertion Sort	"Split Sort"
10000	0.304	0.160	0.161
20000	1.218	0.630	0.387
30000	2.790	1.427	0.726
40000	4.646	2.520	1.285
50000	7.395	4.181	2.719
60000	10.584	5.635	2.897
70000	14.149	8.143	3.939
80000	18.674	10.333	5.079
90000	23.165	12.832	6.375

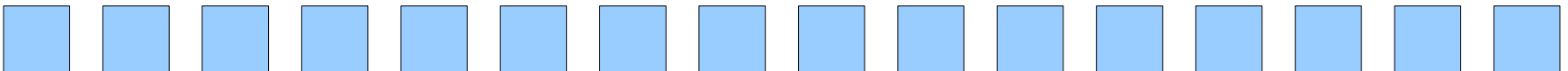
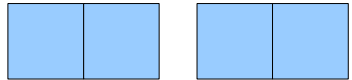
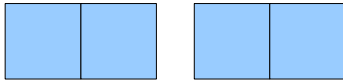
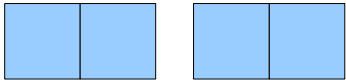
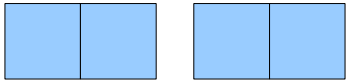
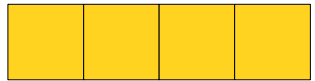
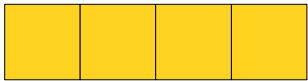
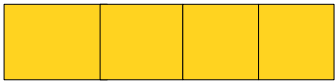
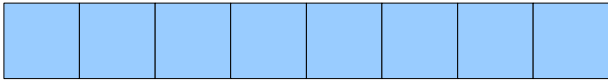
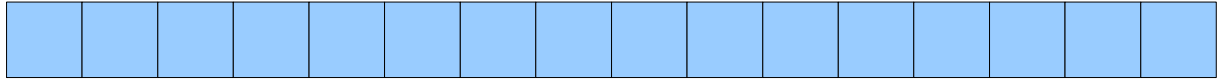
A Better Idea

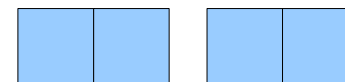
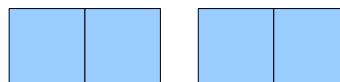
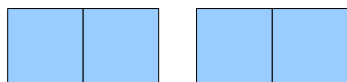
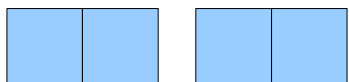
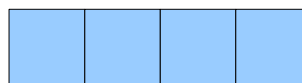
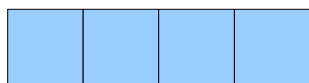
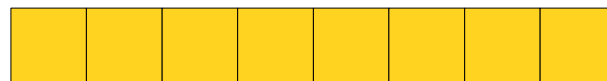
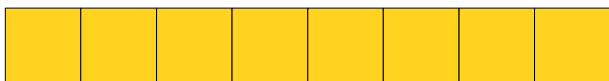
- Splitting the input in half and merging halves the work.
- So why not split into four? Or eight?
- **Question:** What happens if we *never stop splitting*?

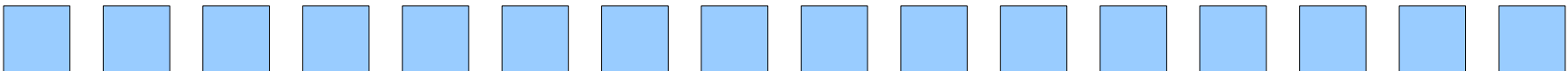
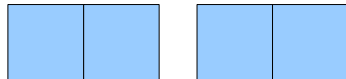
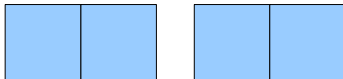
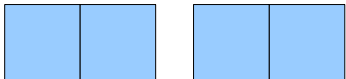
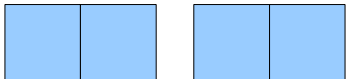
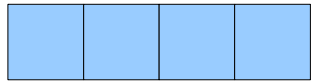
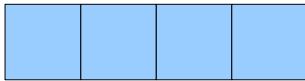
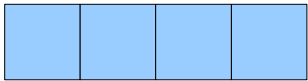
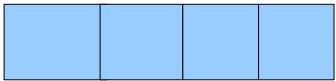
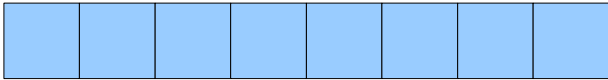












High-Level Idea

- A recursive sorting algorithm!
- Break the list into two halves and recursively sort each.
- Use merge to combine them back into a single sorted list.
- This algorithm is called *mergesort*.
- Questions:
 - What does this look like in code?
 - How efficient is it?

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

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

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

What is the runtime 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.add(v[i]);
    for (int i = v.size() / 2; i < v.size(); i++)
        right.add(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.add(v[i]);
    for (int i = v.size() / 2; i < v.size(); i++)
        right.add(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.add(v[i]);
    for (int i = v.size() / 2; i < v.size(); i++)
        right.add(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.add(v[i]);
    for (int i = v.size() / 2; i < v.size(); i++)
        right.add(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.add(v[i]);
    for (int i = v.size() / 2; i < v.size(); i++)
        right.add(v[i]);

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

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

How do we
analyze this step?

Recurrence Relations

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n$$

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n \\ &= 4T(n/4) + 2n \\ &= 4(2T(n/8) + n/4) + 2n \\ &= 8T(n/8) + 3n \\ &\dots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$

Recurrence Relations

$$T(1) = 1$$

$$T(n) = 2^k T(n / 2^k) + kn$$

What if $n / 2^k = 1$?
Then $T(n) = 2^k + kn$

$$n / 2^k = 1$$

$$n = 2^k$$

$$\log_2 n = k$$

So

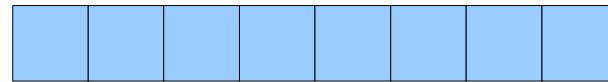
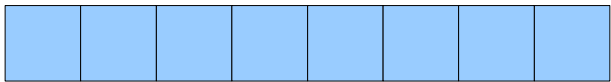
$$T(n) = n + n \log_2 n$$

$$= \mathbf{O(n \log n)}$$

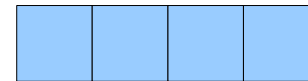
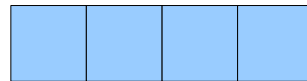
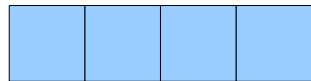
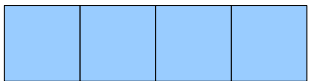
A Graphical Proof



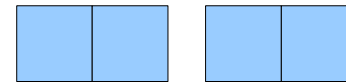
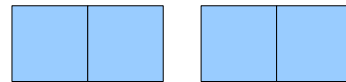
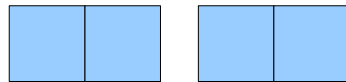
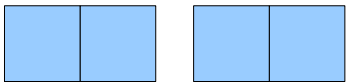
$O(n)$



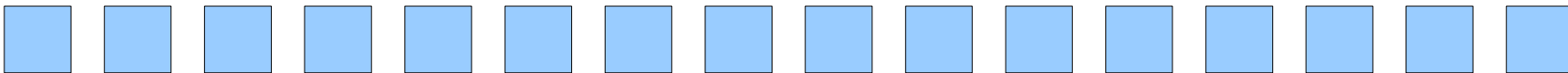
$O(n)$



$O(n)$



$O(n)$



$O(n)$

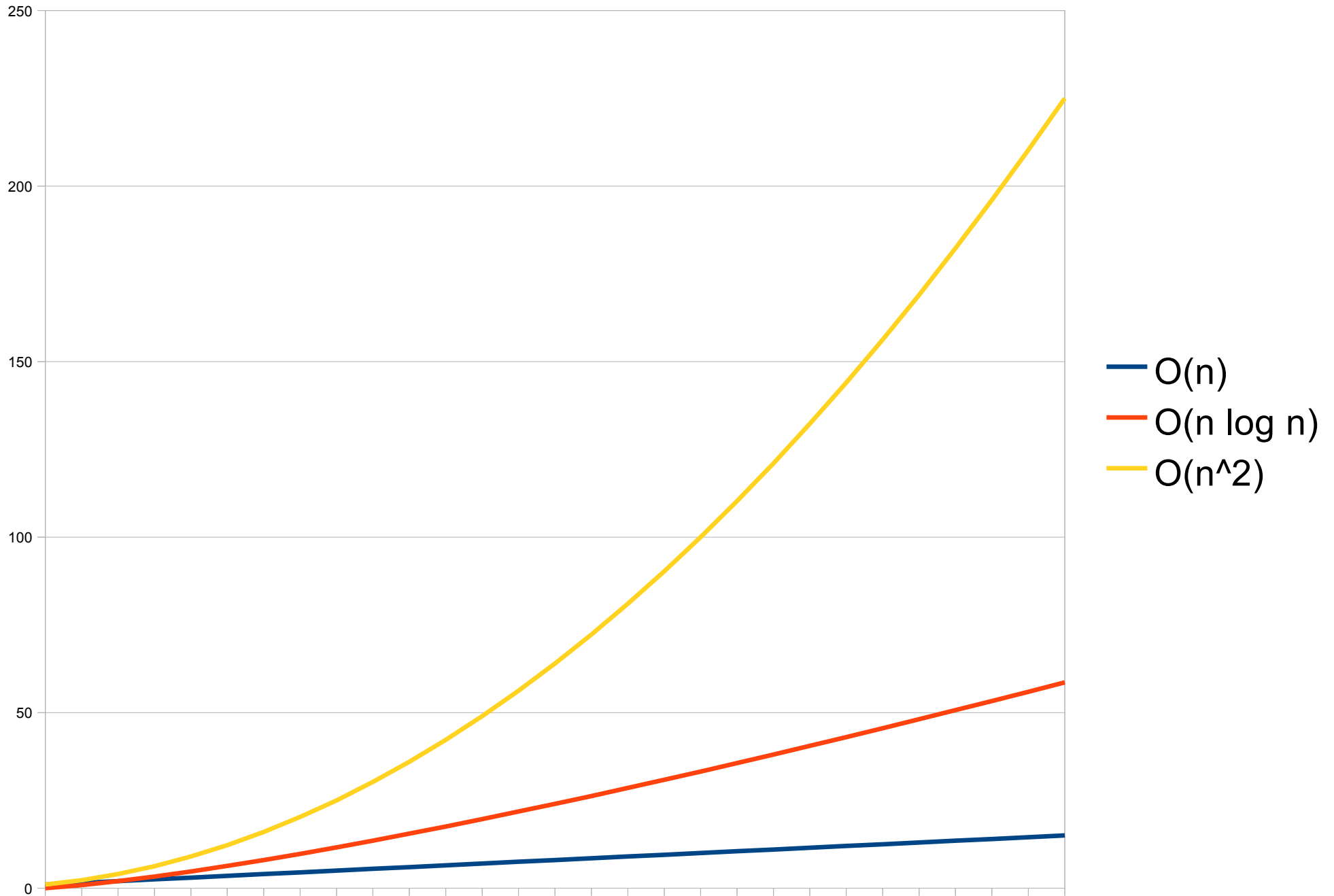
Analysis of Mergesort

- Mergesort is $O(n \log n)$.
- This is asymptotically better than $O(n^2)$
- Mergesort is *much* faster than insertion sort or selection sort.

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

Growth Rates



A Note on $O(n \log n)$

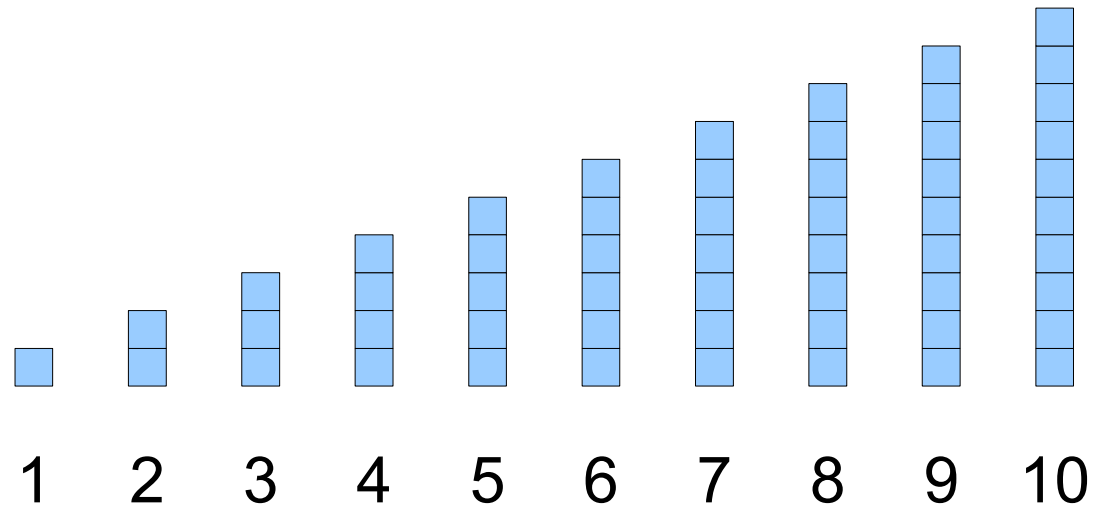
- No need to specify the base of the logarithm.
- Why?
 - $\log_b n = \log_a n / \log_a b$
 - All logarithms are constant multiples of one another.
- New notation: $\lg n$ means $\log_2 n$

Can we do better than $O(n \log n)$?

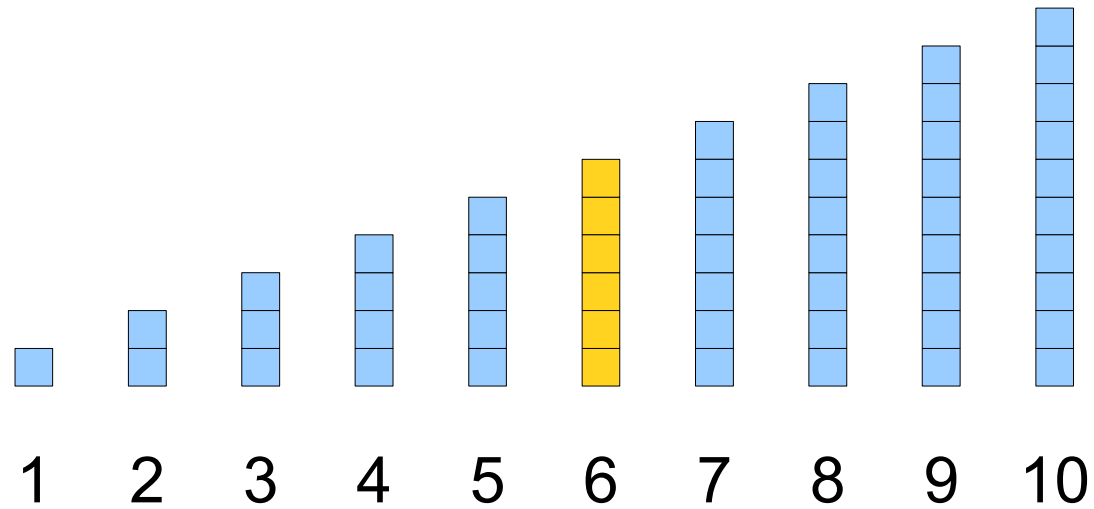
- We were able to improve on insertion sort's and selection sort's $O(n^2)$.
- Can we do better than $O(n \log n)$?
- In general, **no**:
 - Result from information theory: No comparison sort can do better than $O(n \log n)$.
- **No comparison sort has a better big-O than mergesort!**

A Trivial Observation

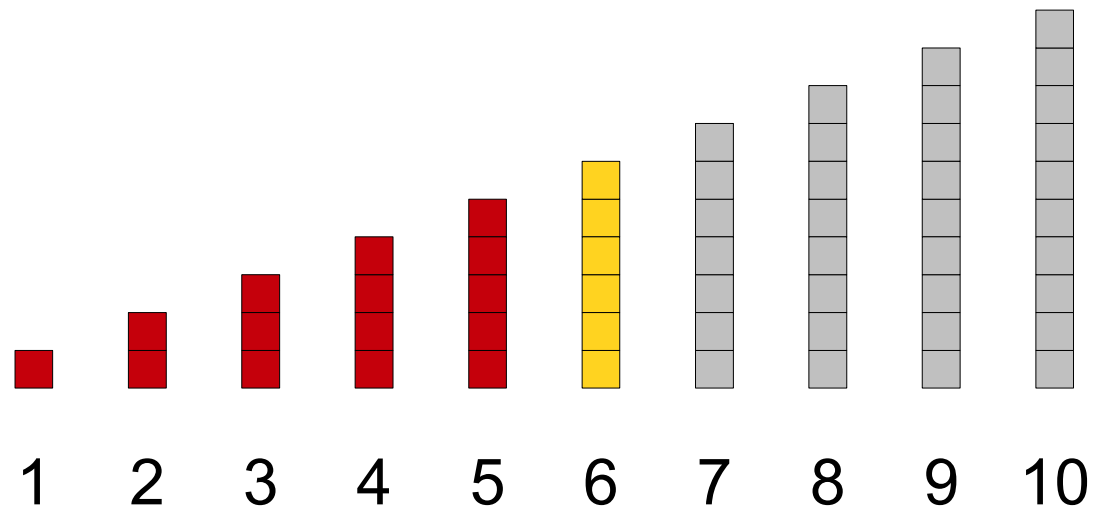
A Trivial Observation



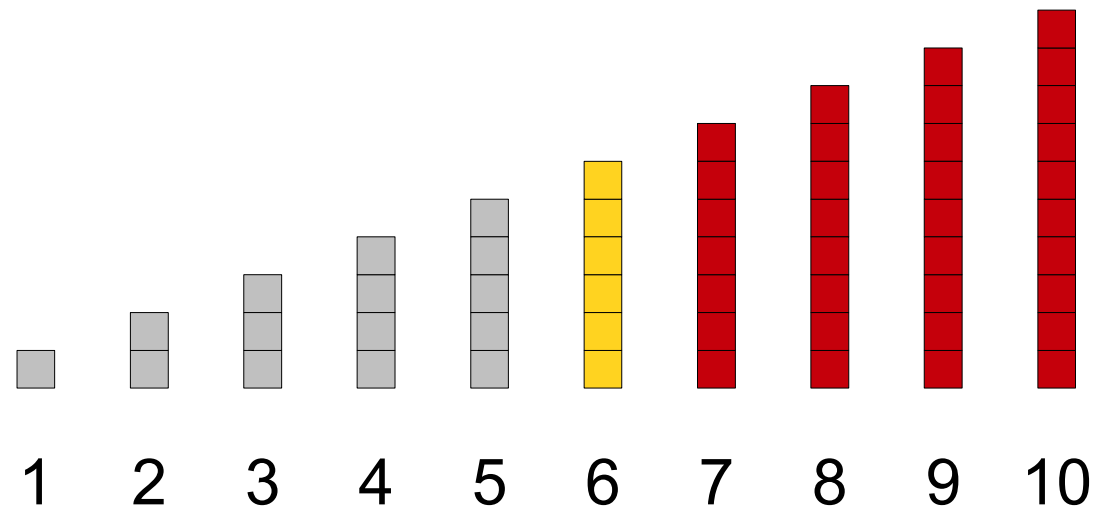
A Trivial Observation



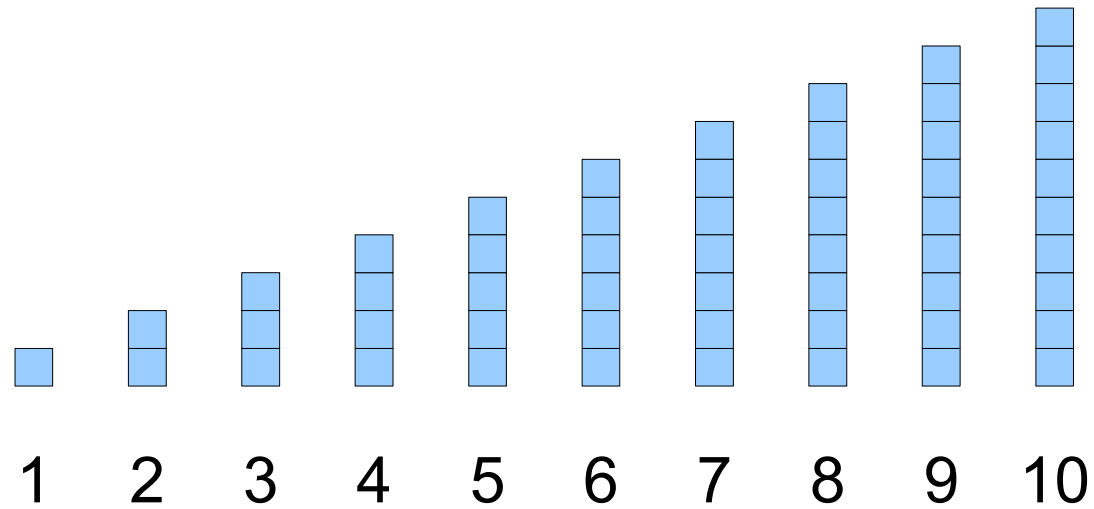
A Trivial Observation



A Trivial Observation



A Trivial Observation



So What?

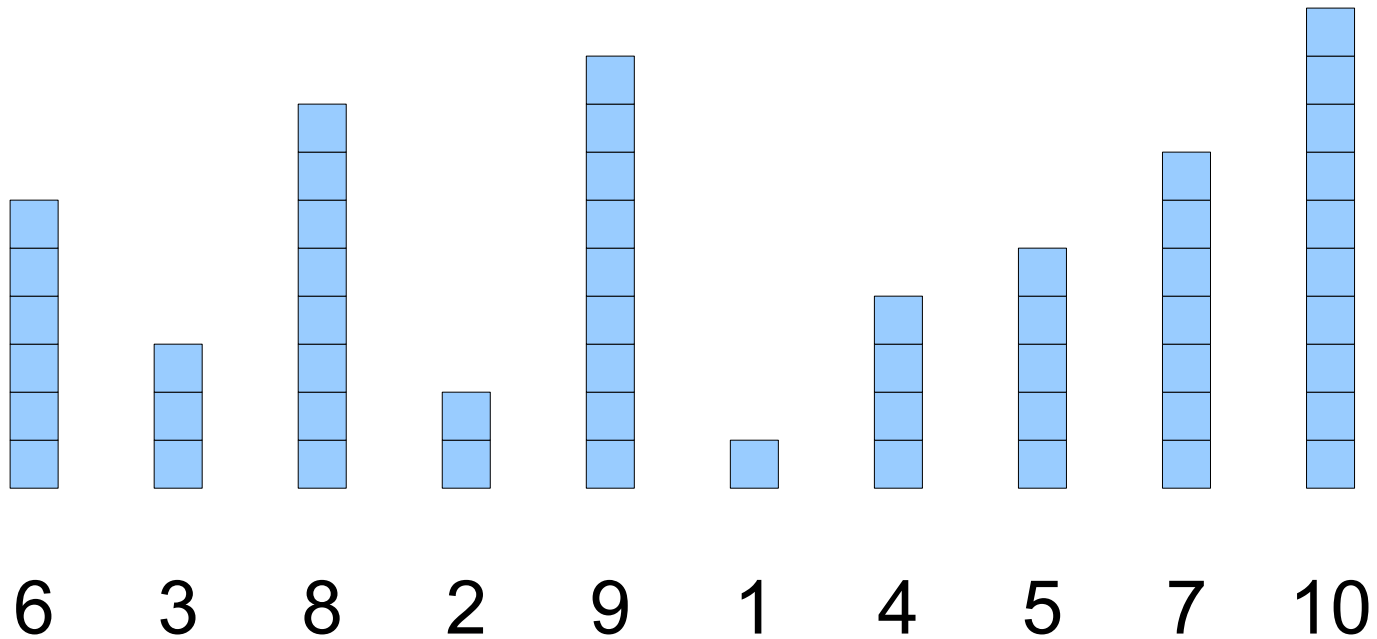
- This idea leads to a particularly clever sorting algorithm.
- Idea:
 - Pick an element from the array.
 - Put the smaller elements on one side.
 - Put the bigger elements on the other side.
 - Recursively sort each half.
- But how do we do the middle two steps?

Partitioning

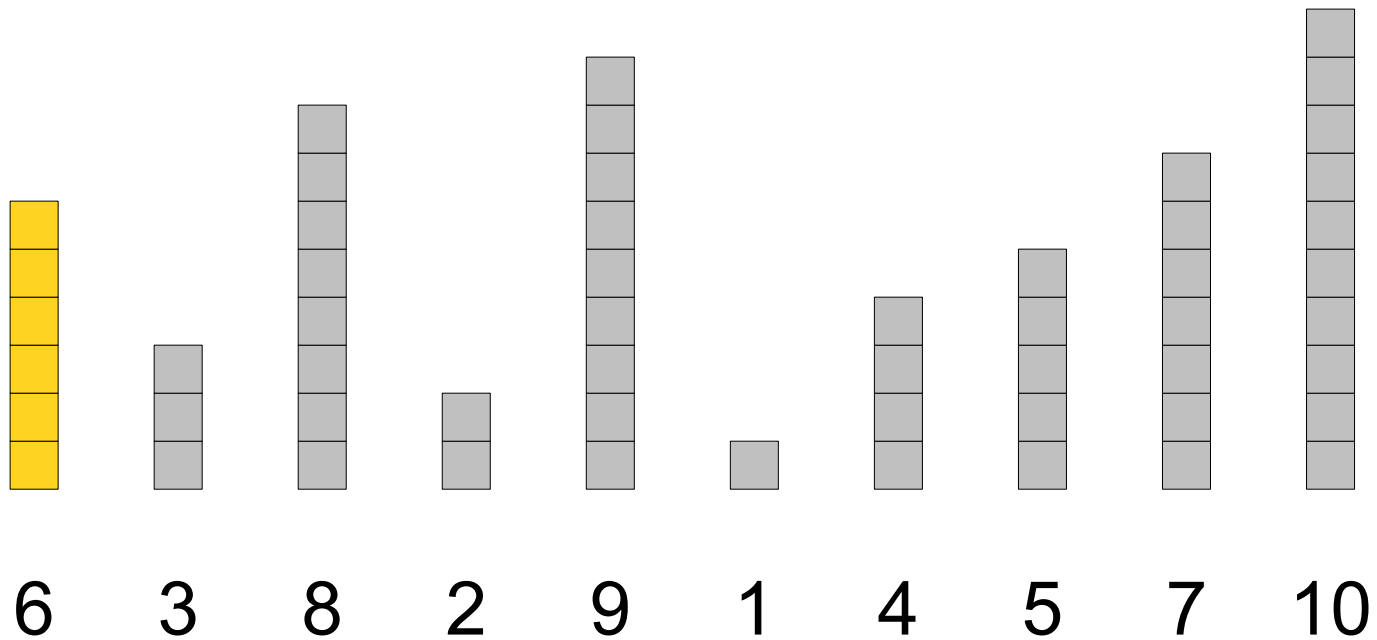
- Pick a **pivot element**.
- Move everything less than the pivot to the left of the pivot.
- Move everything greater than the pivot to the right of the pivot.
- Good news: $O(n)$ algorithm exists!
- Bad news: it's a bit tricky...

The Partition Algorithm

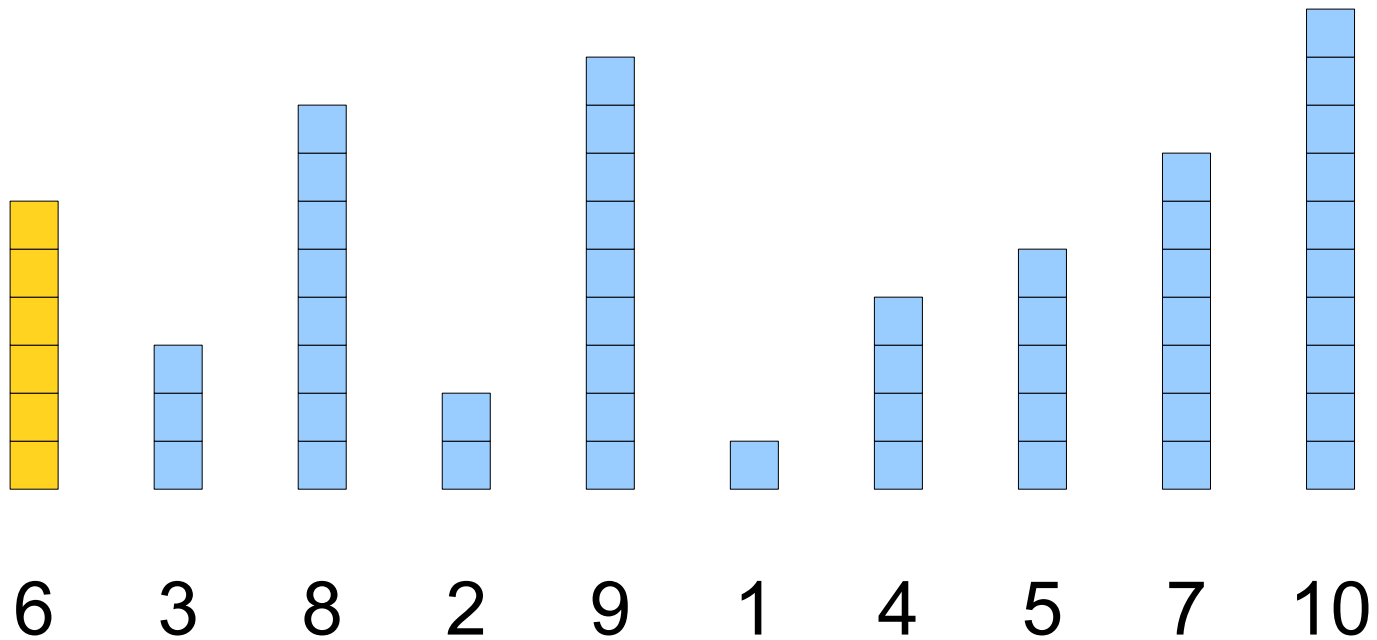
The Partition Algorithm



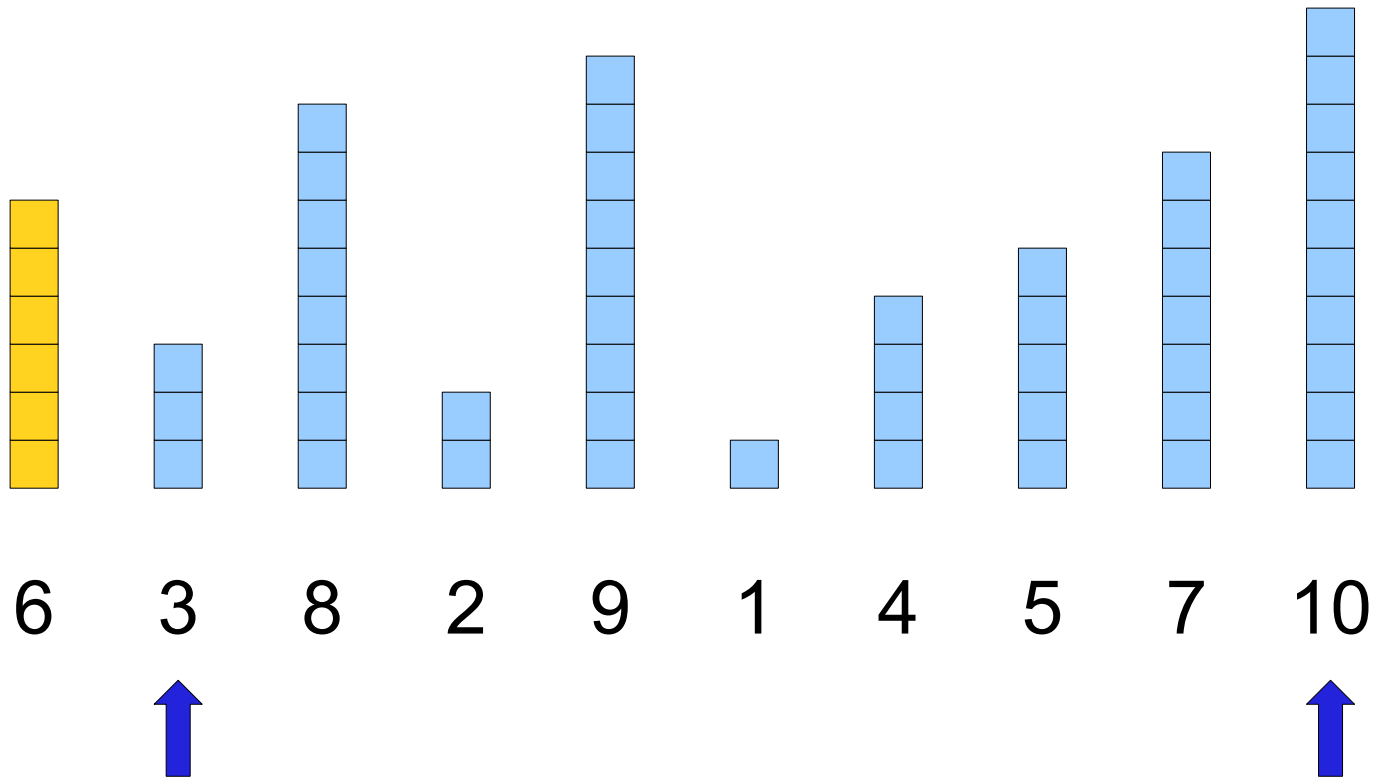
The Partition Algorithm



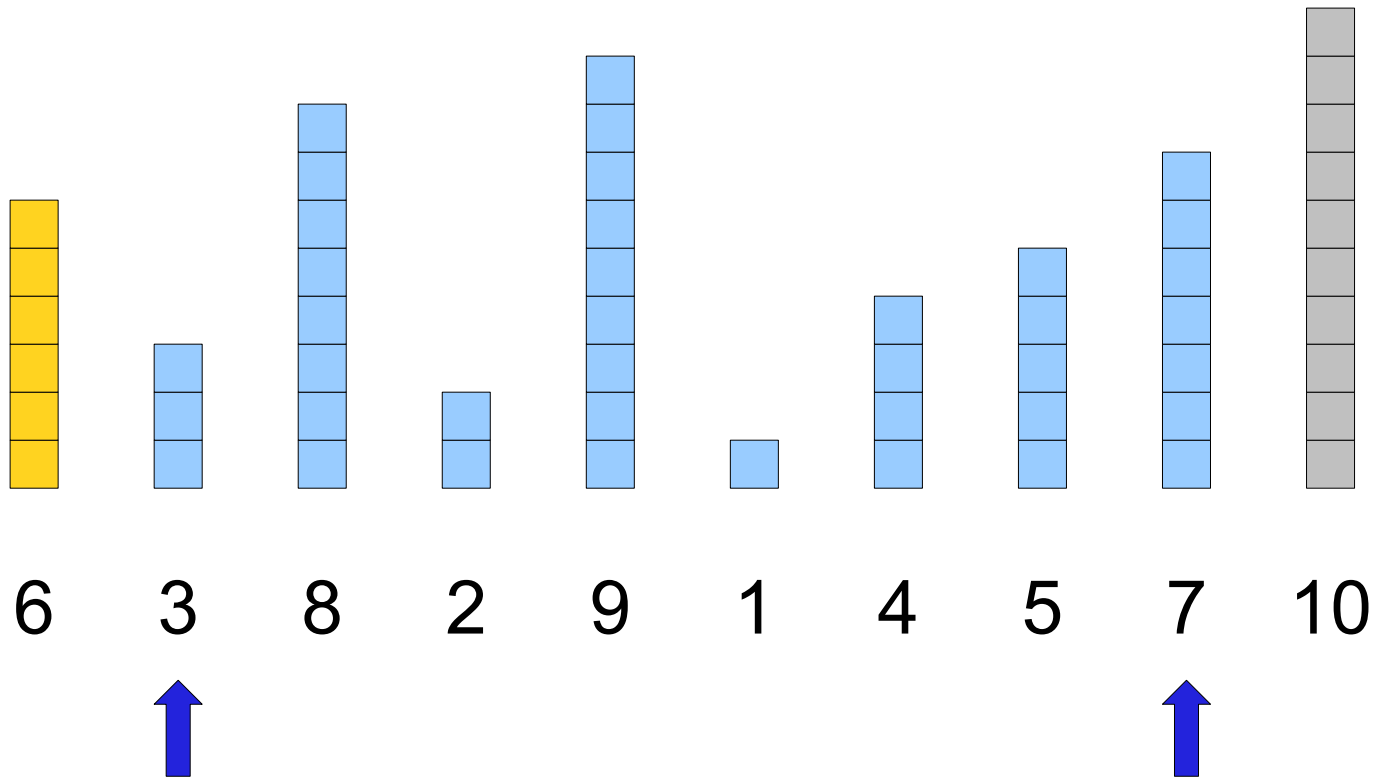
The Partition Algorithm



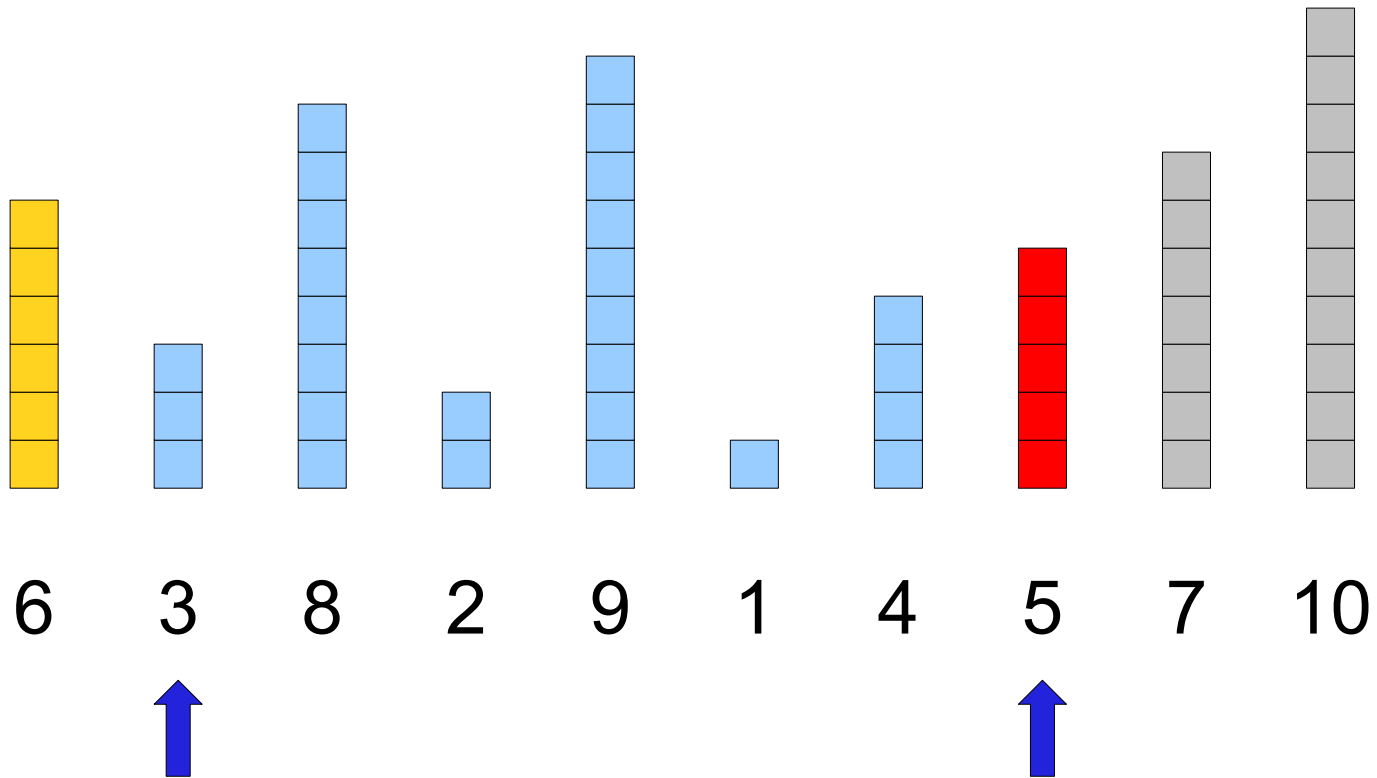
The Partition Algorithm



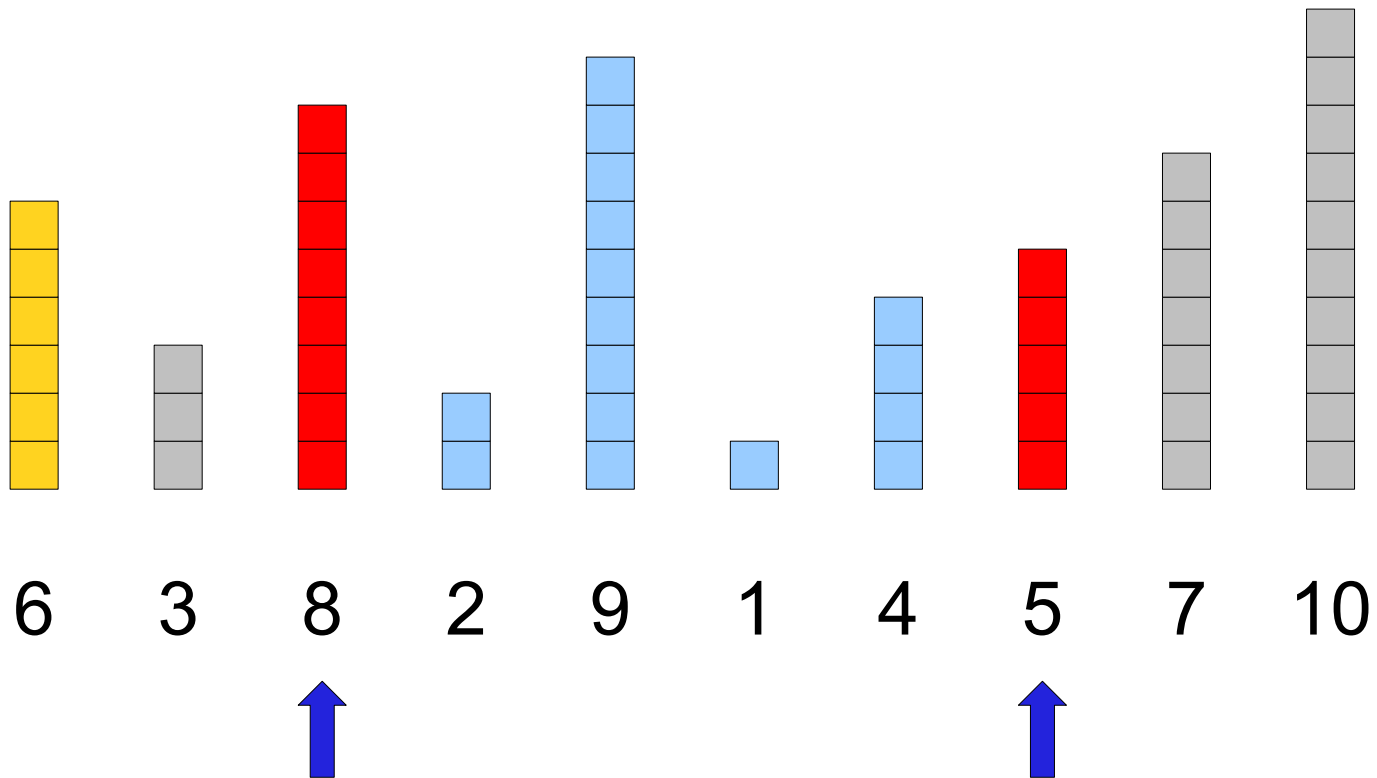
The Partition Algorithm



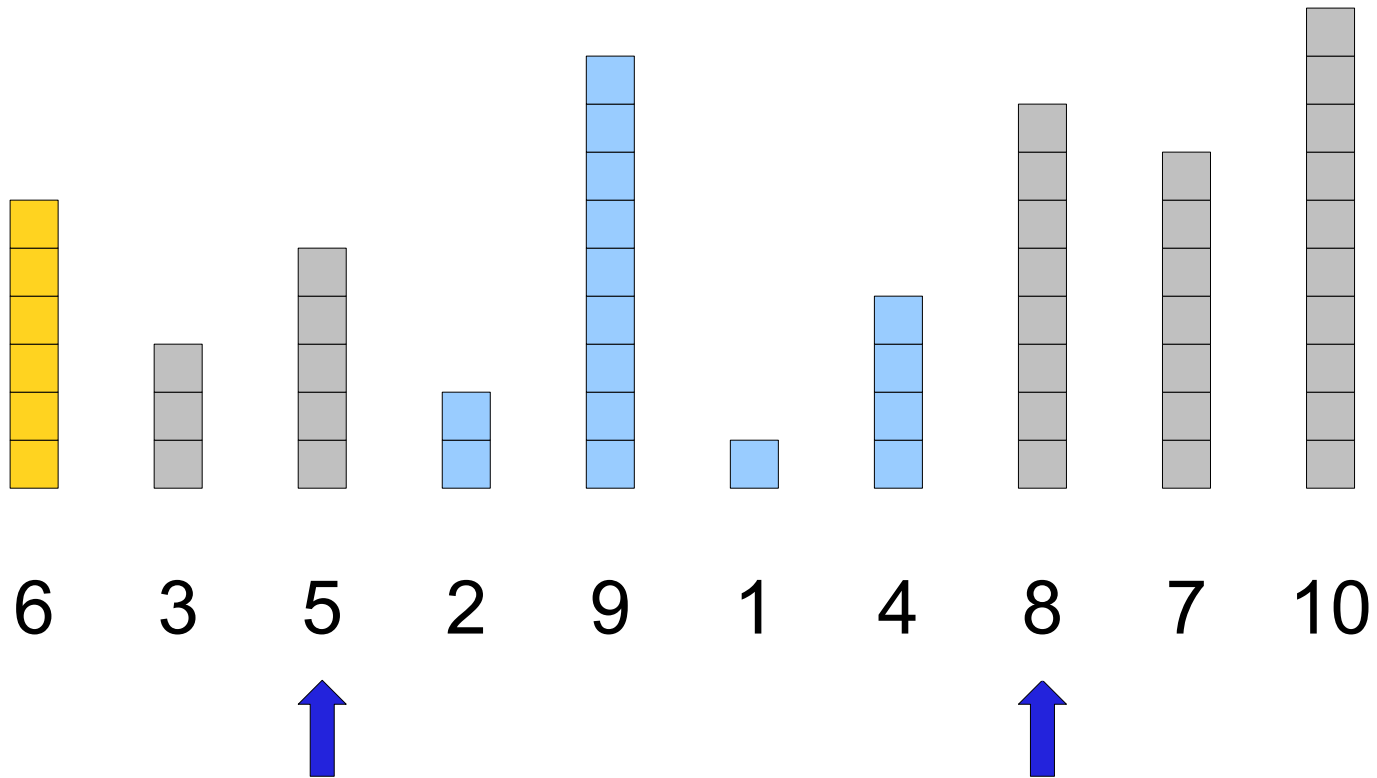
The Partition Algorithm



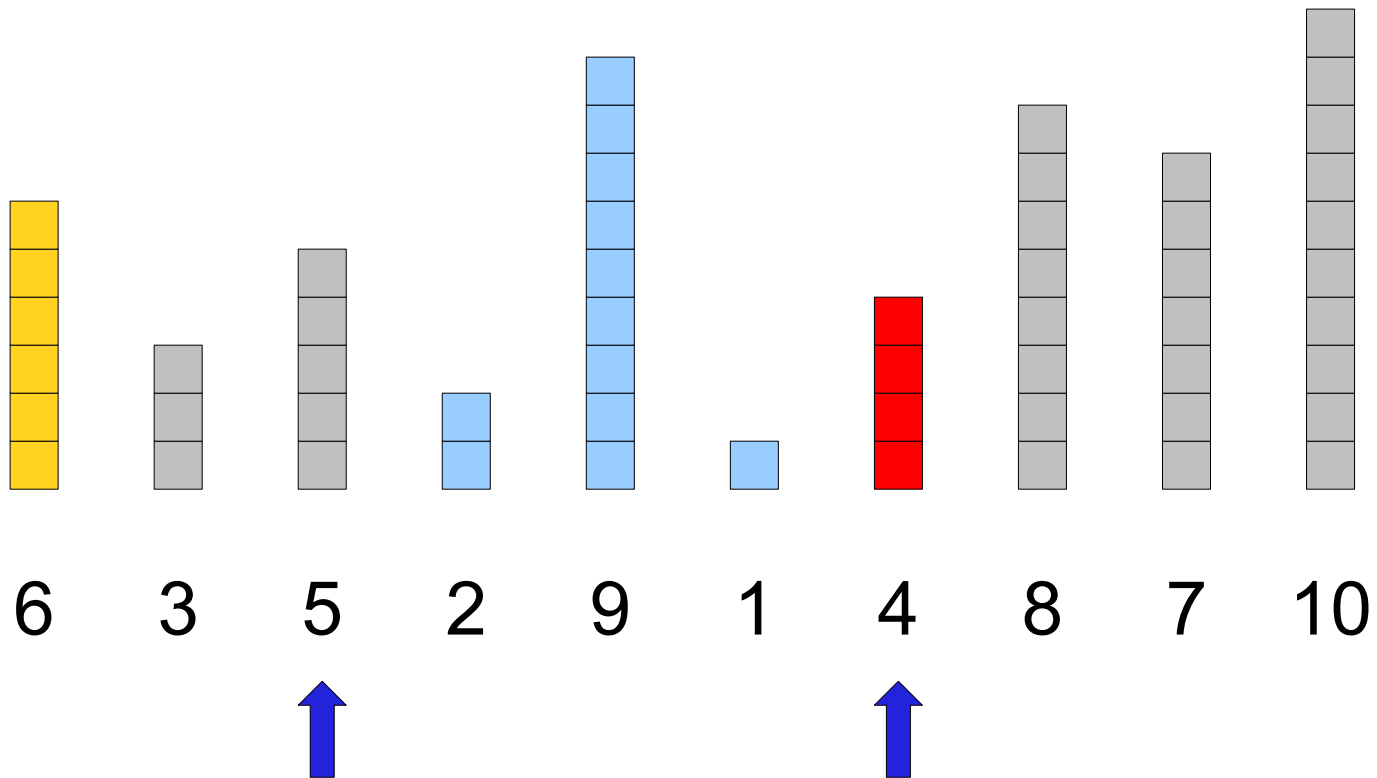
The Partition Algorithm



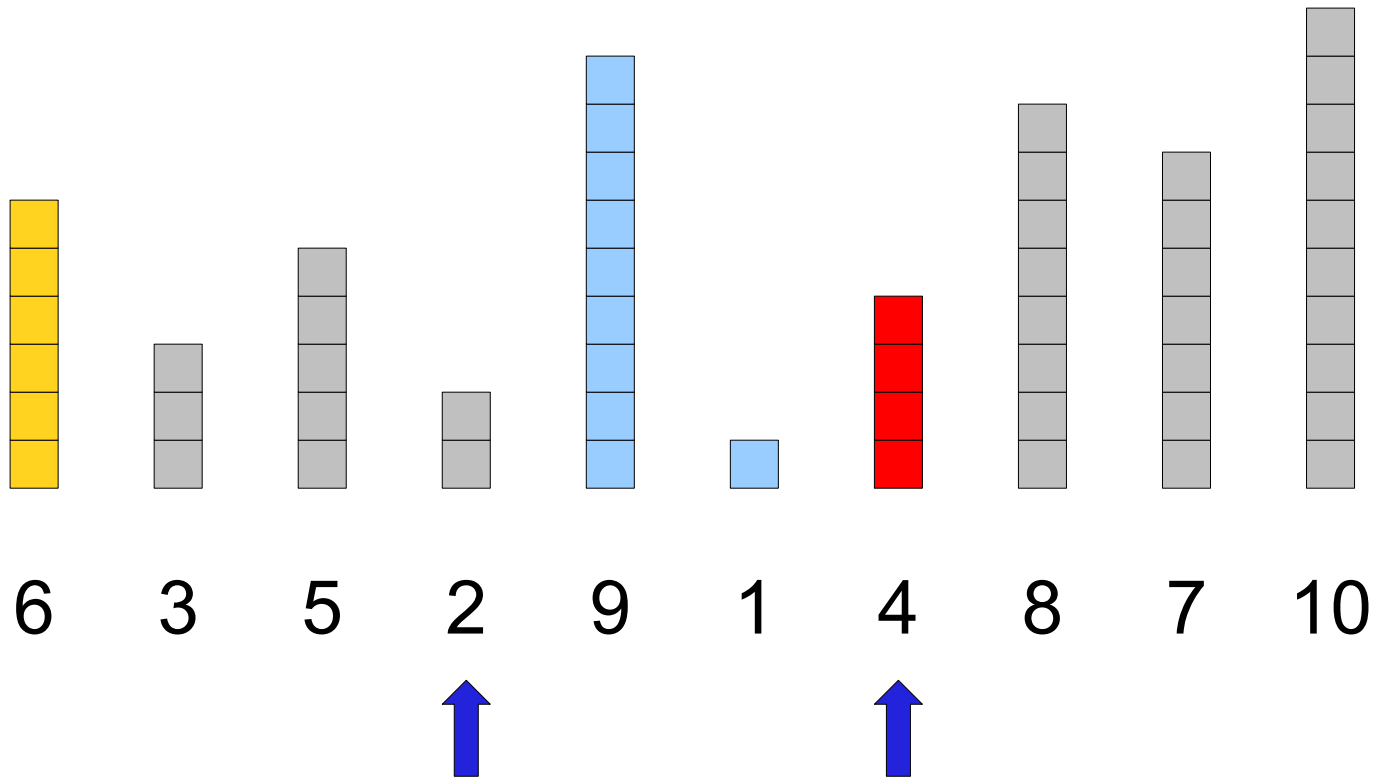
The Partition Algorithm



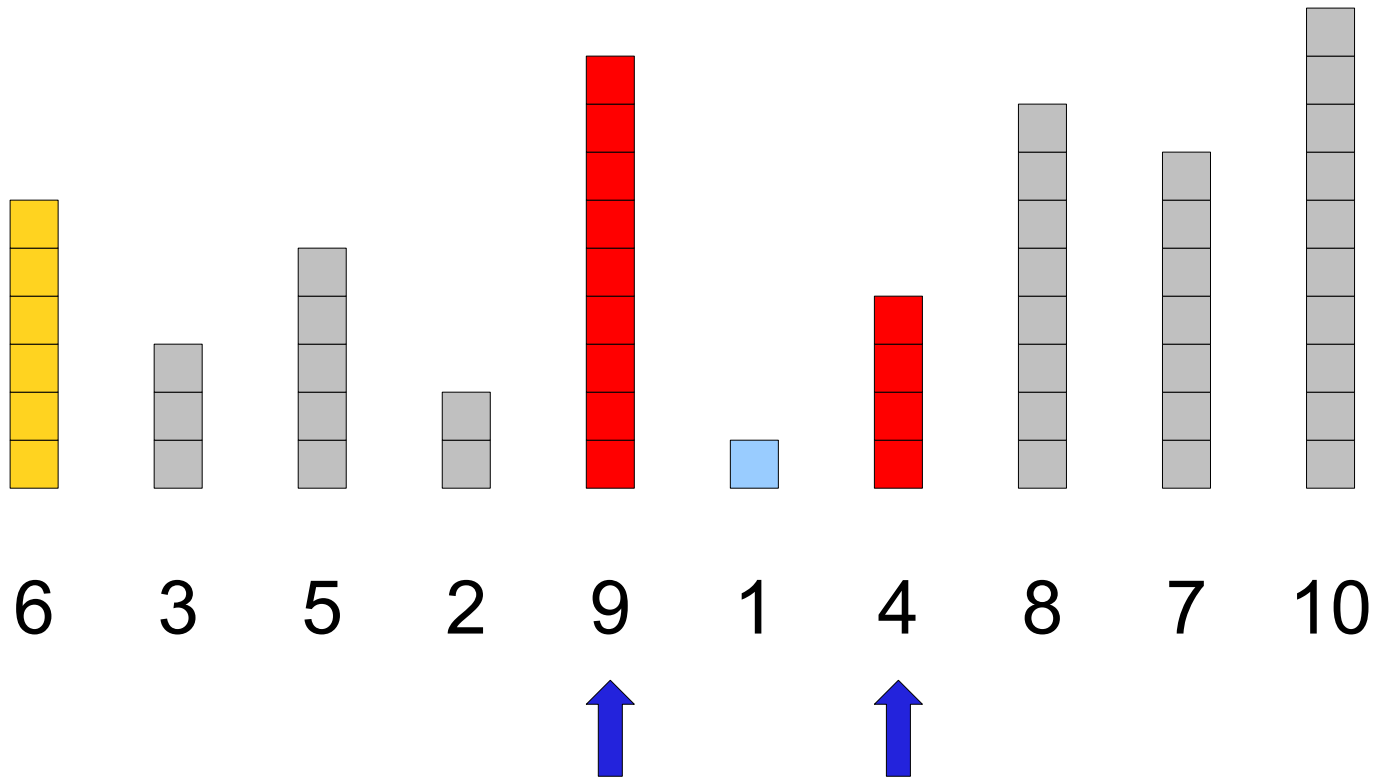
The Partition Algorithm



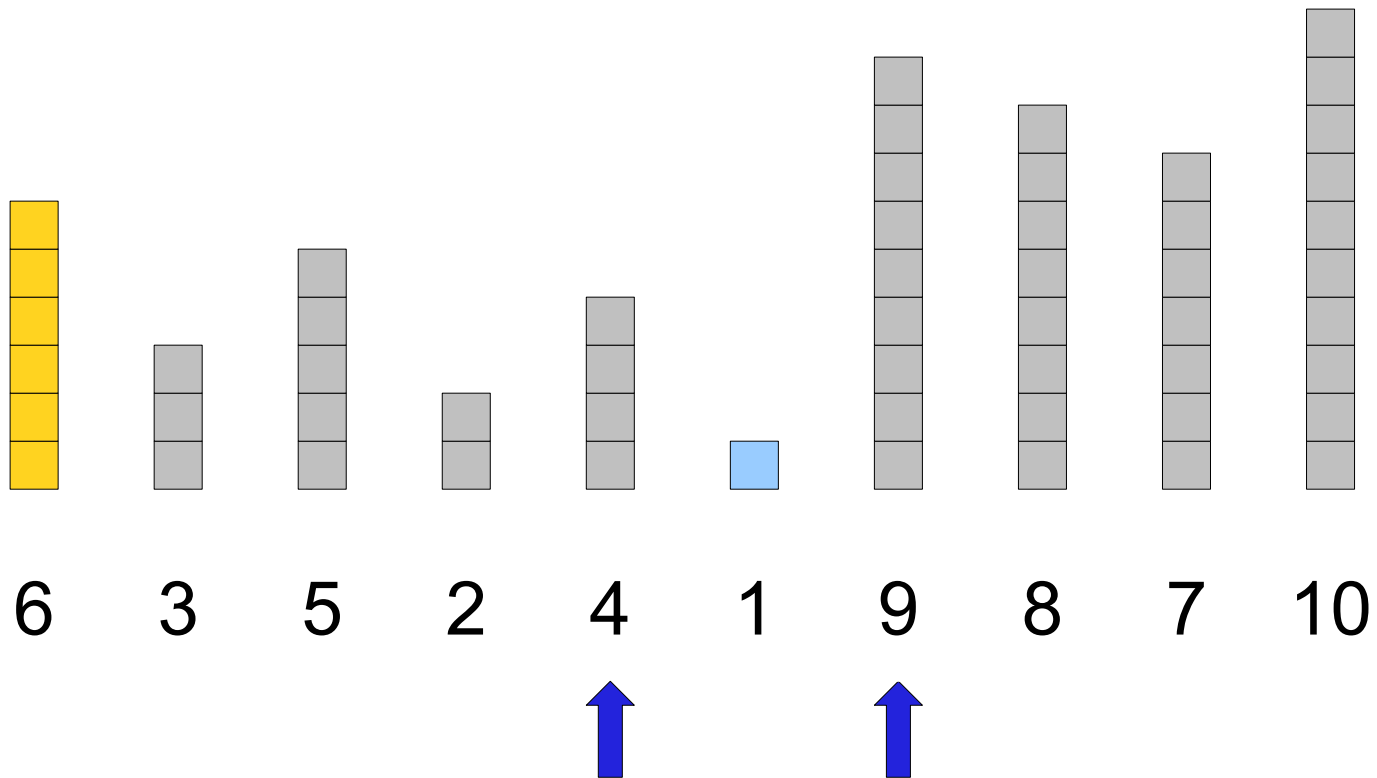
The Partition Algorithm



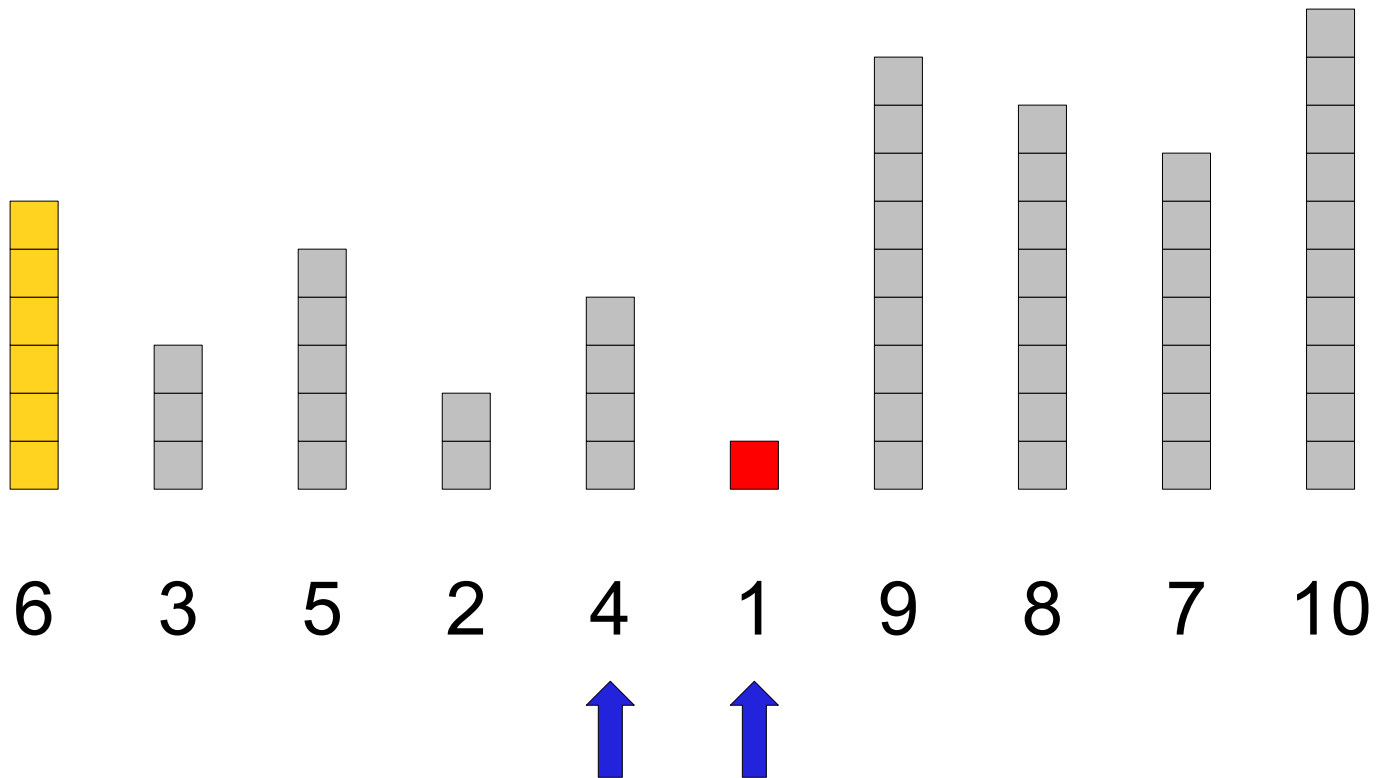
The Partition Algorithm



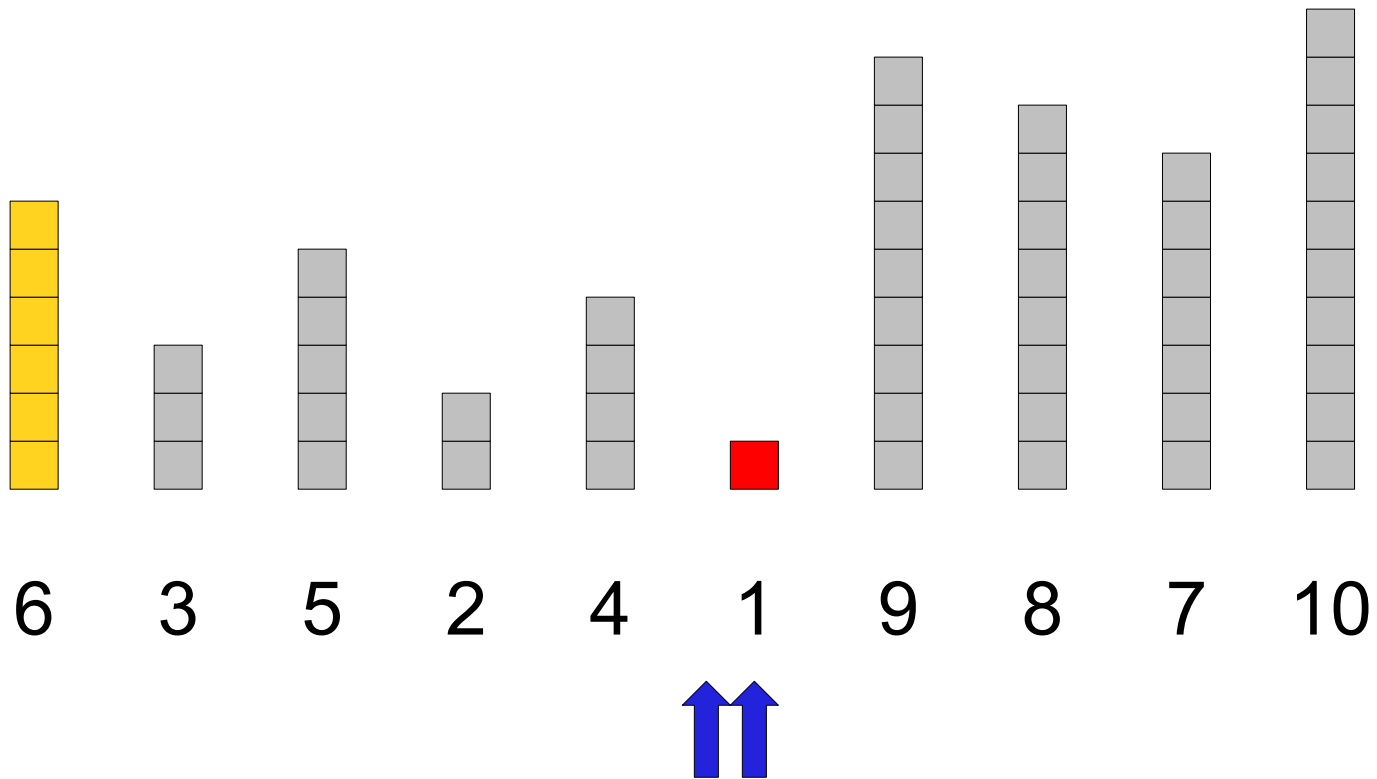
The Partition Algorithm



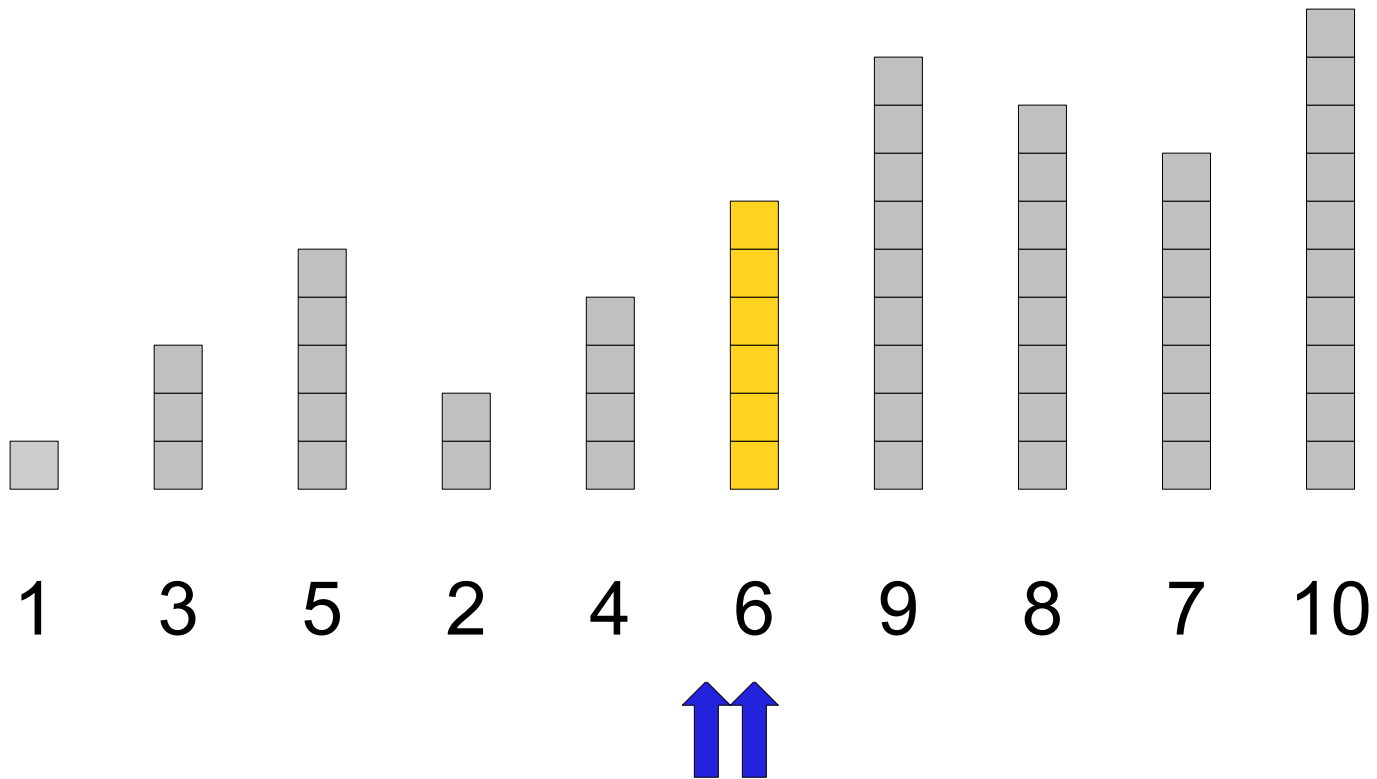
The Partition Algorithm



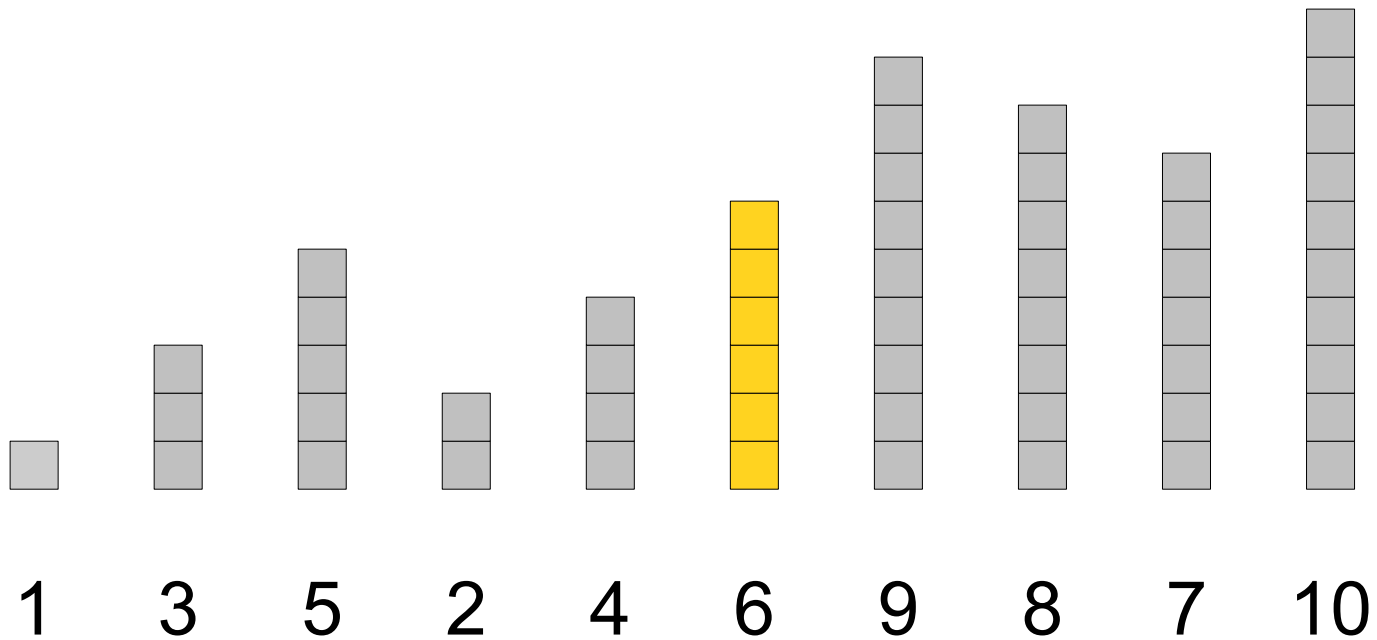
The Partition Algorithm



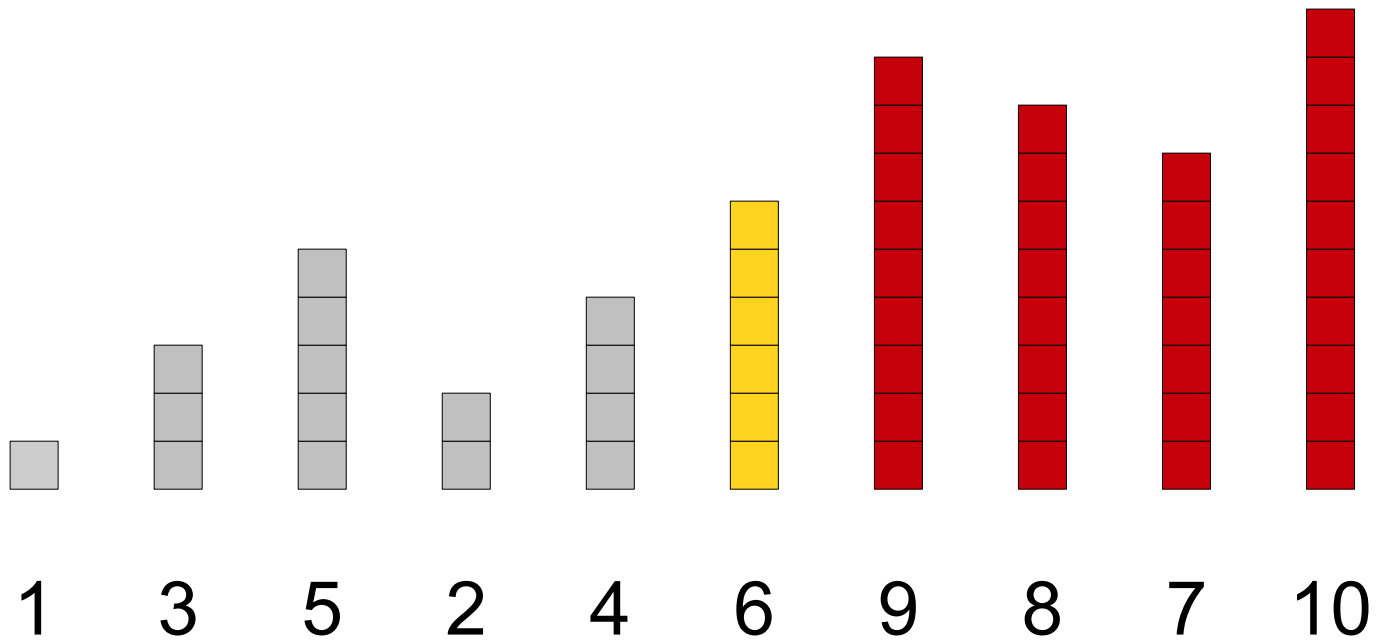
The Partition Algorithm



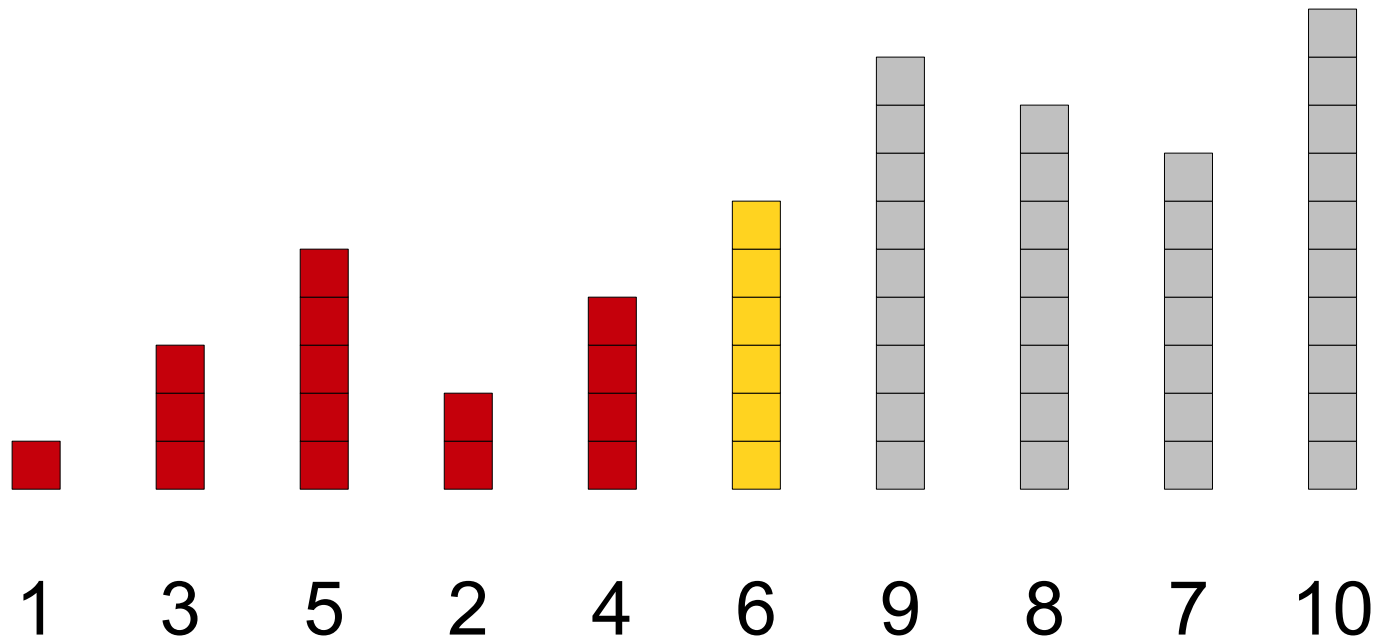
The Partition Algorithm



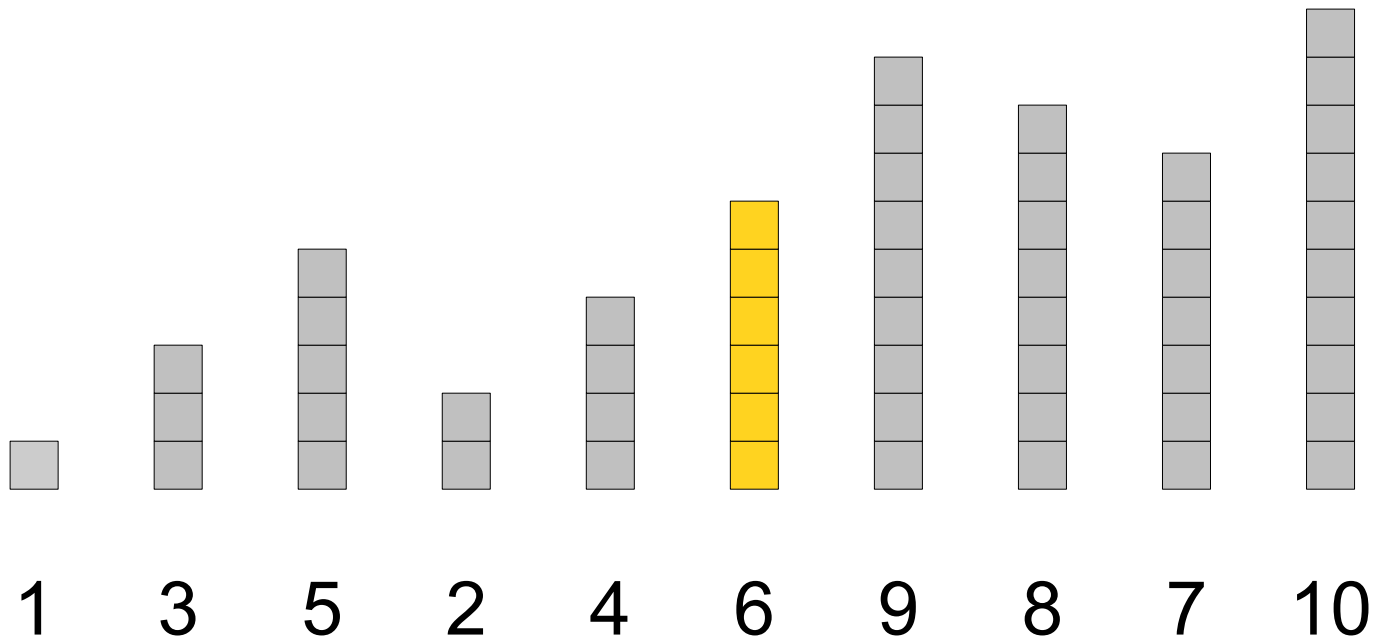
The Partition Algorithm



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

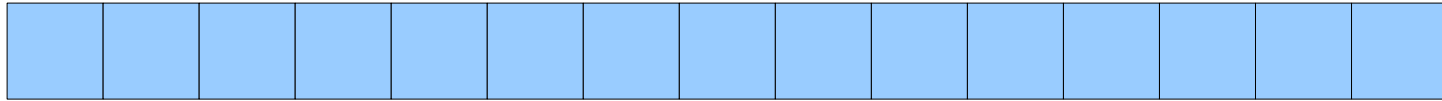
```
void Quicksort(Vector<int>& v, int low, int high)
{
    if (low >= high) return;

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

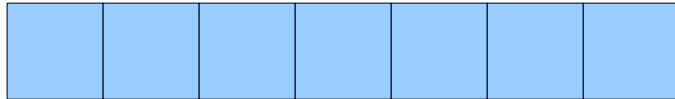
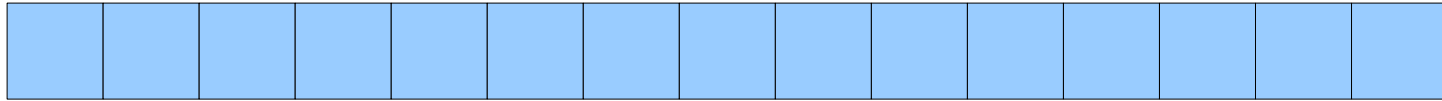
How fast is quicksort?

It depends on our choice of pivot.

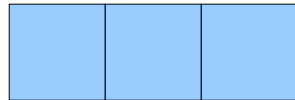
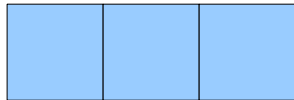
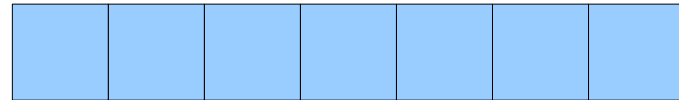
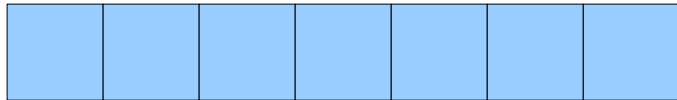
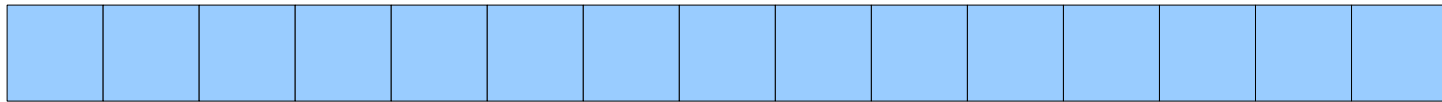
Suppose we get lucky...



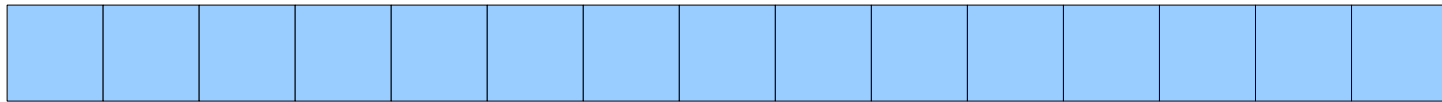
Suppose we get lucky...



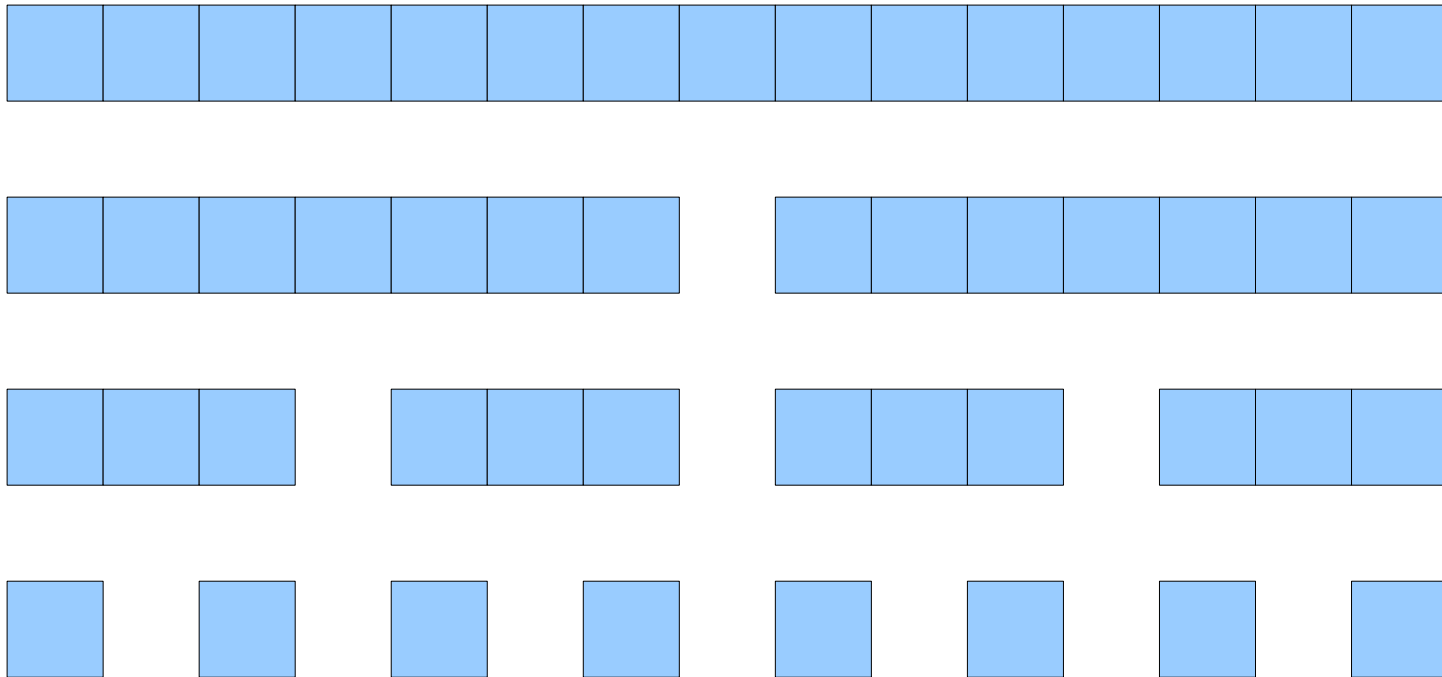
Suppose we get lucky...



Suppose we get lucky...



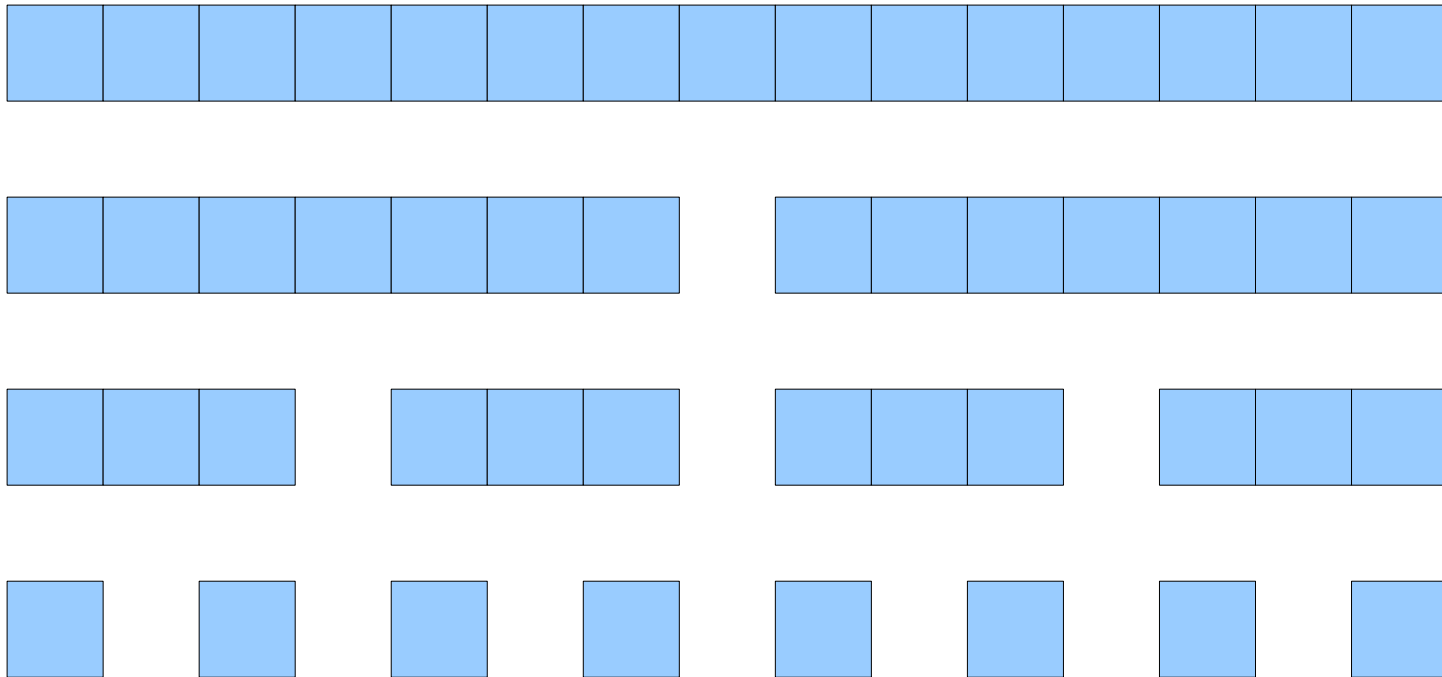
Suppose we get lucky...



$$T(1) = 1$$

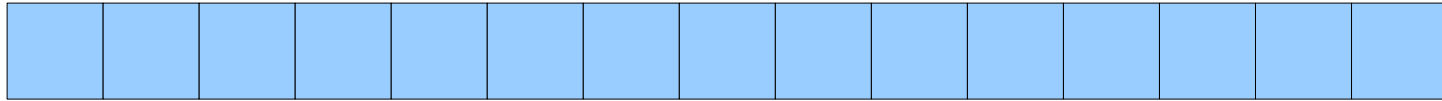
$$T(n) = 2T(n / 2) + n$$

Suppose we get lucky...

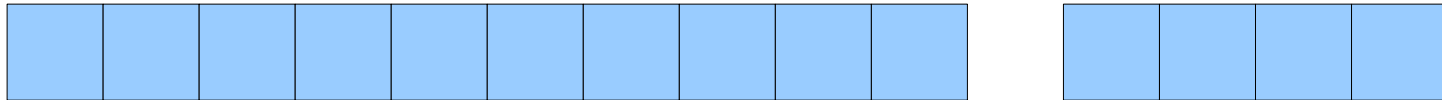
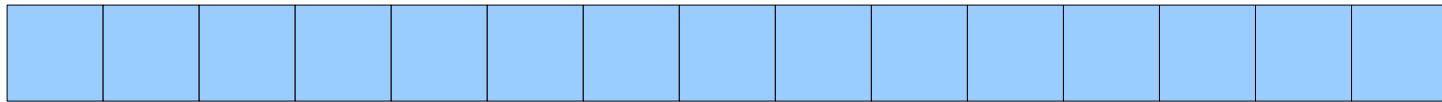


$O(n \log n)$

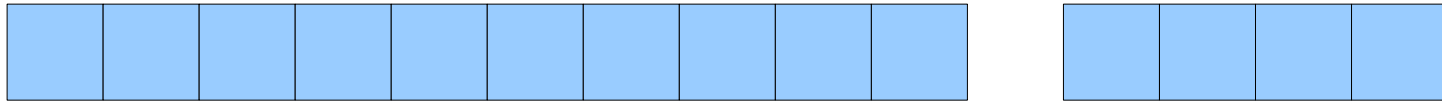
Suppose we get *sorta* lucky...



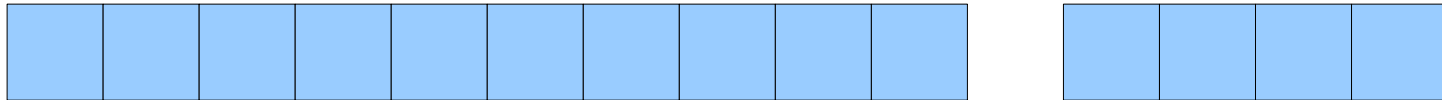
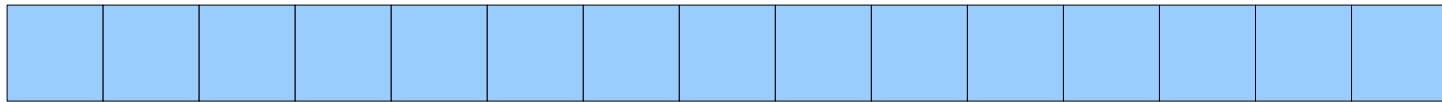
Suppose we get *sorta* lucky...



Suppose we get *sorta* lucky...



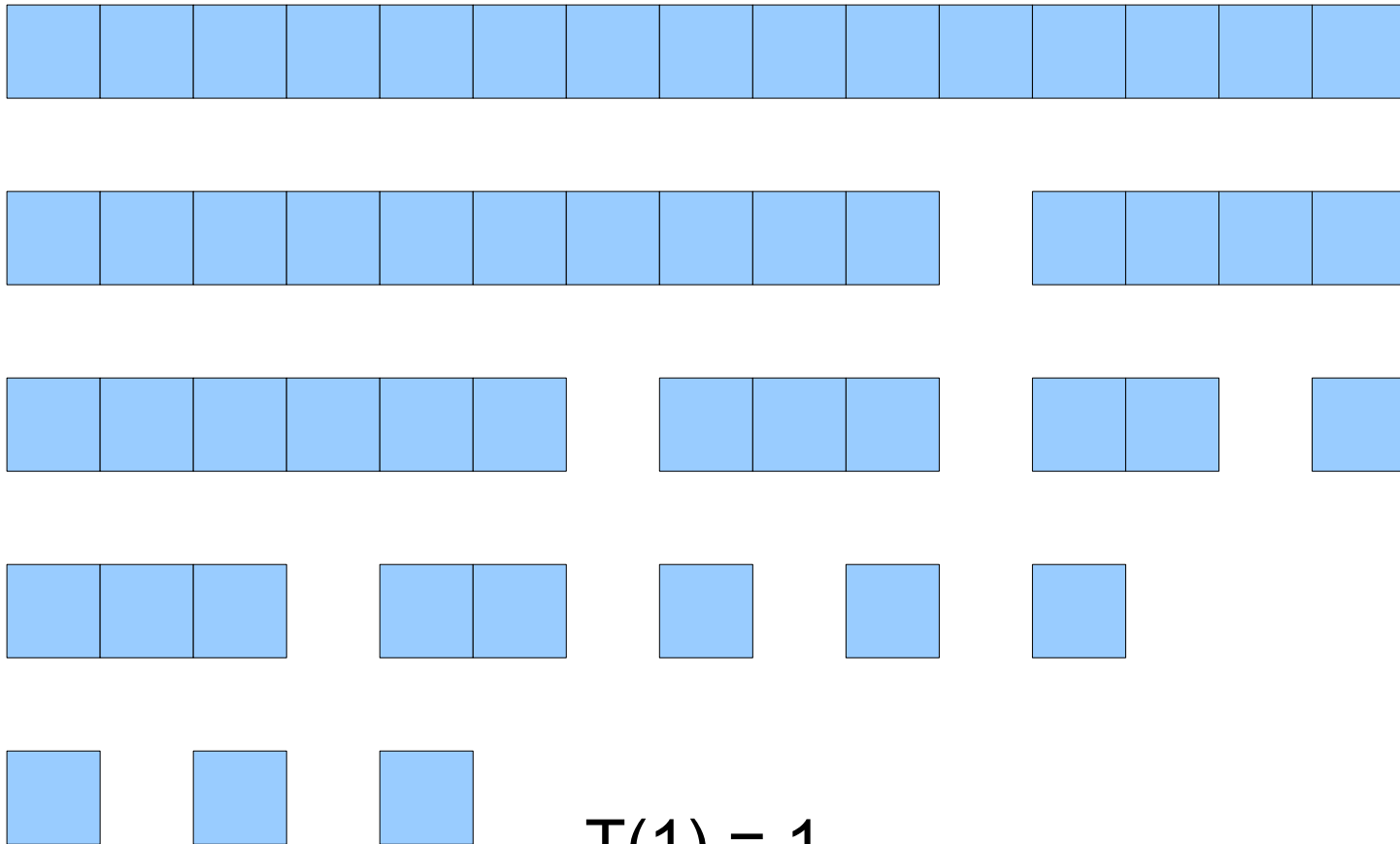
Suppose we get *sorta* lucky...



Suppose we get *sorta* lucky...



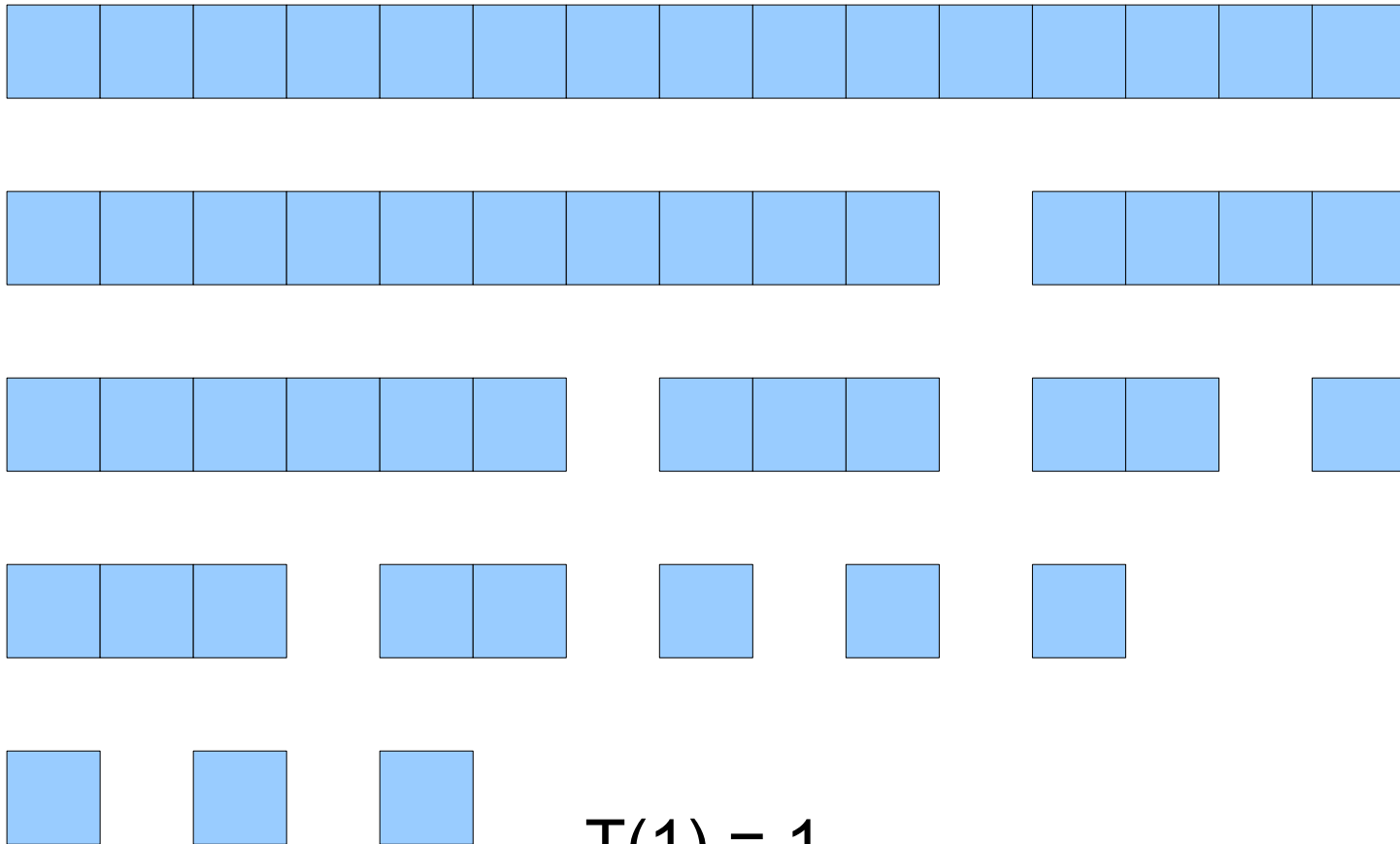
Suppose we get *sorta* lucky...



$$T(1) = 1$$

$$T(n) = T(7n / 10) + T(3n / 10) + n$$

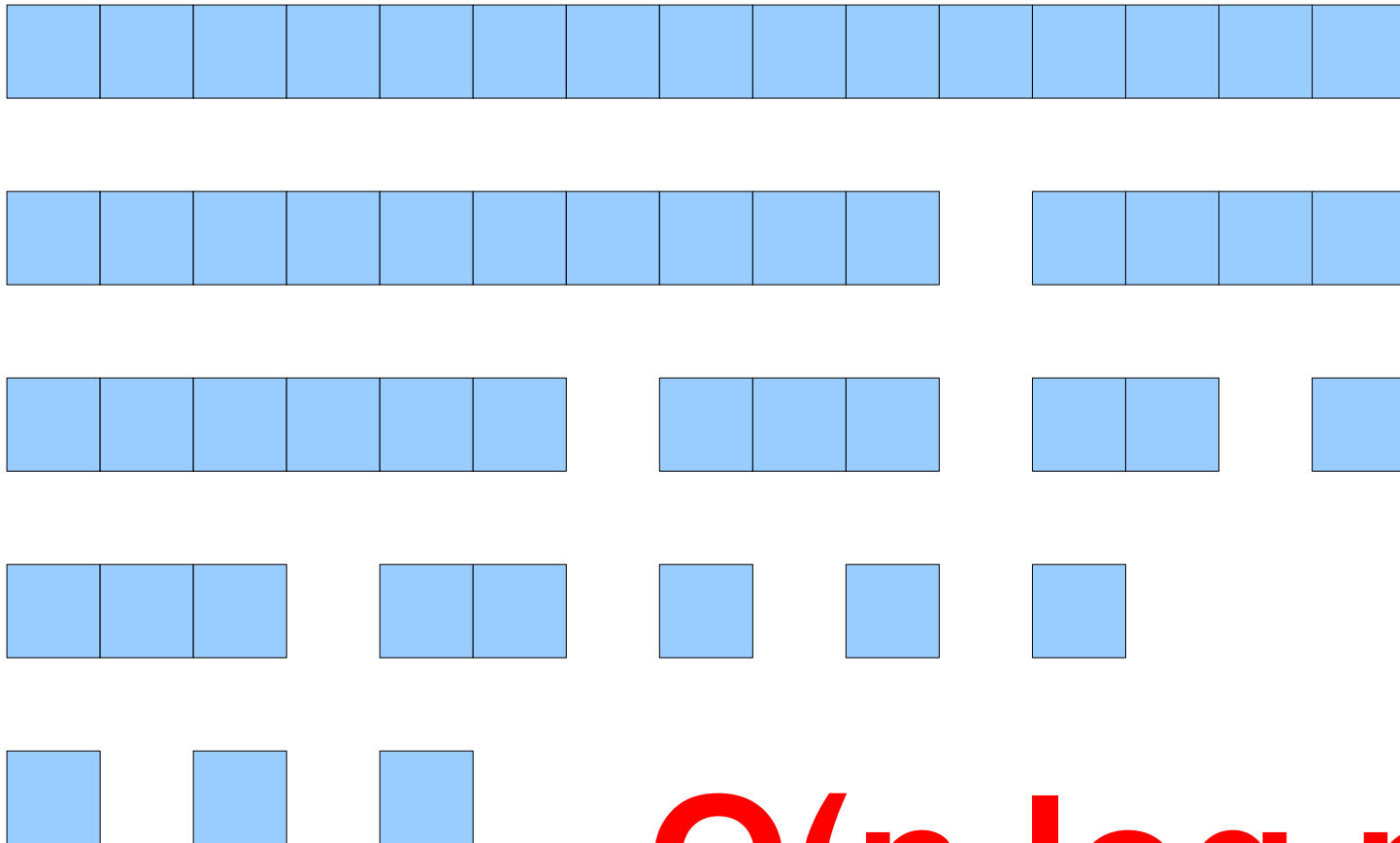
Suppose we get *sorta* lucky...



$$T(1) = 1$$

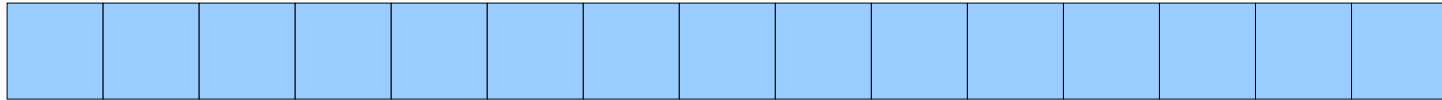
$$T(n) = T(7n / 10) + T(3n / 10) + n$$

Suppose we get *sorta* lucky...

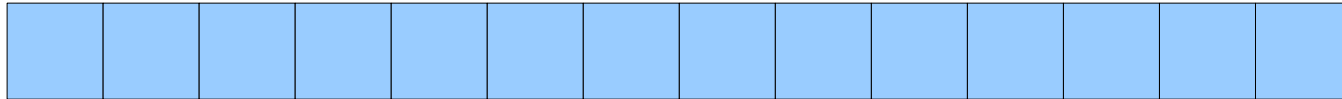
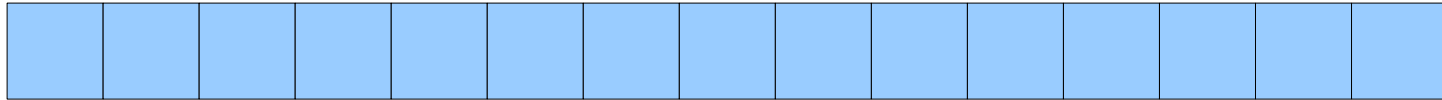


$O(n \log n)$

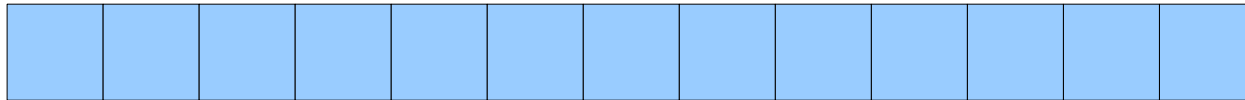
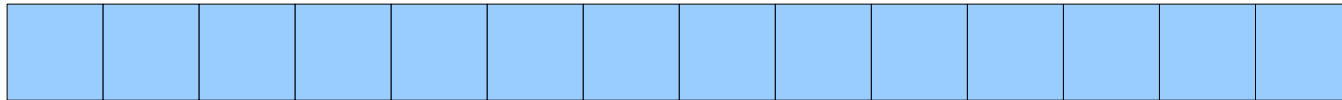
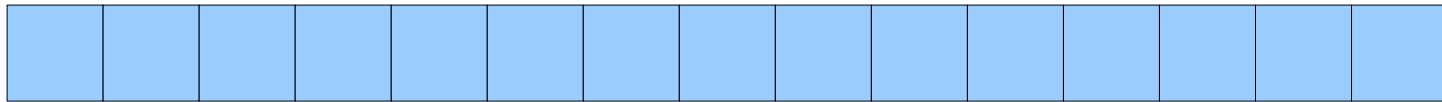
Suppose we get **unlucky**



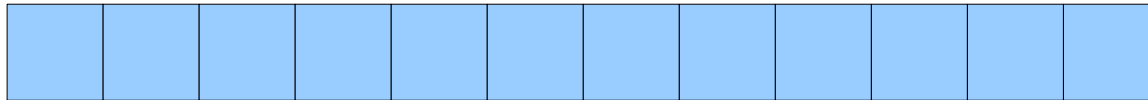
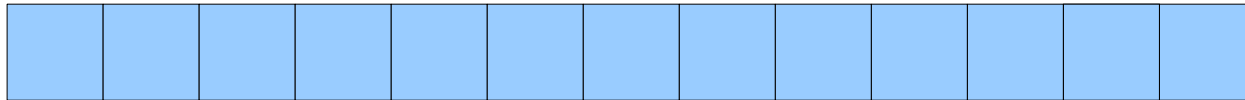
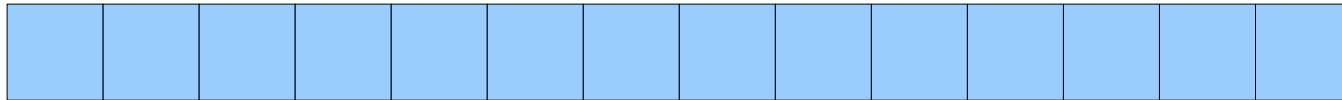
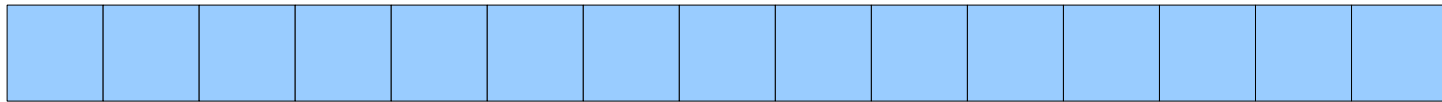
Suppose we get **unlucky**



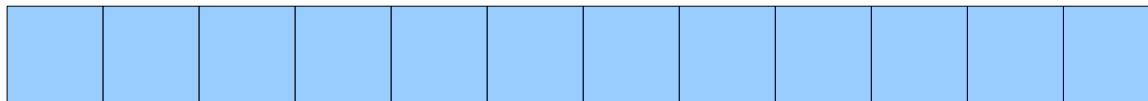
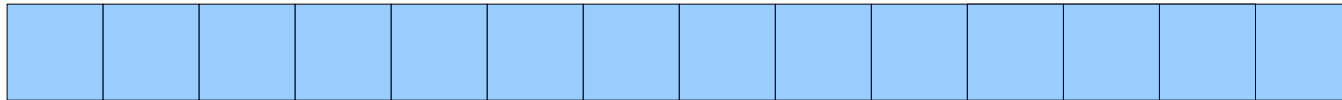
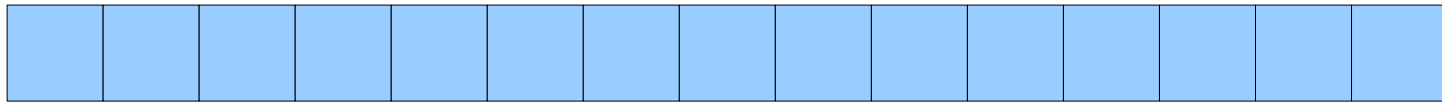
Suppose we get **unlucky**



Suppose we get **unlucky**



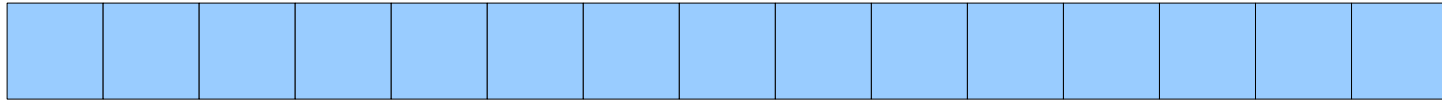
Suppose we get **unlucky**



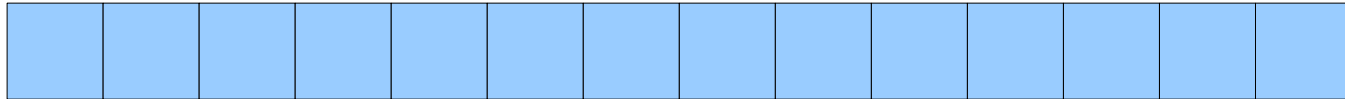
...

Suppose we get **unlucky**

n



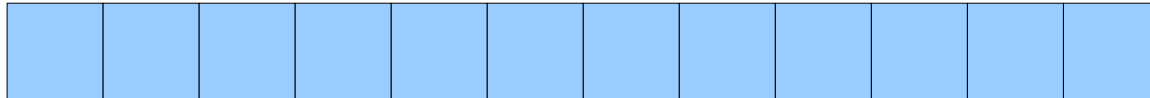
n - 1



n - 2



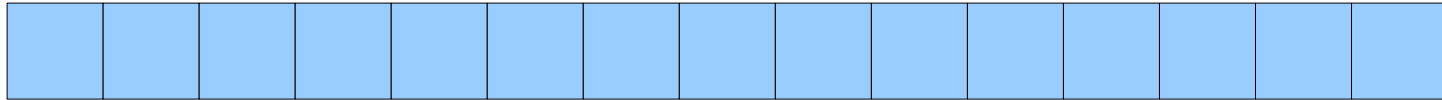
n - 3



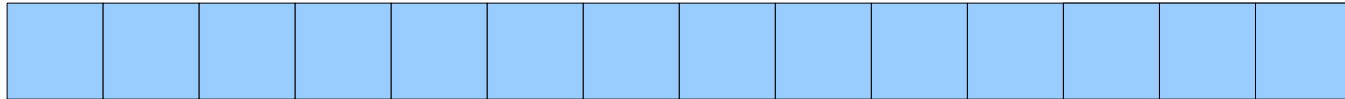
...

Suppose we get **unlucky**

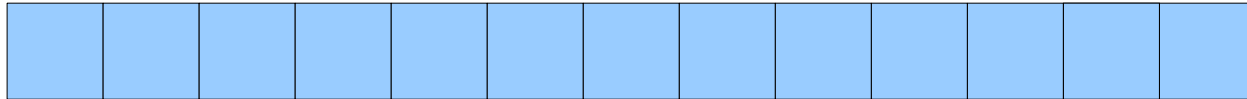
n



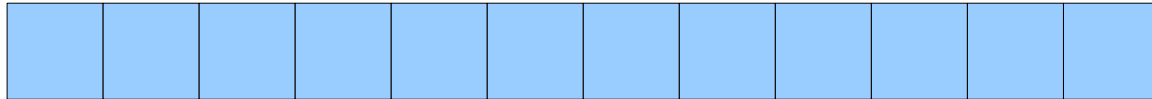
n - 1



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 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	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 is **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?
- **Yes!**

Hybrid Mergesort

Hybrid Mergesort

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

    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++)
        left.add(v[i]);
    for (int i = v.size() / 2; i < v.size(); i++)
        right.add(v[i]);

    HybridMergesort(left);
    HybridMergesort(right);

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

Hybrid Mergesort

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

    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++)
        left.add(v[i]);
    for (int i = v.size() / 2; i < v.size(); i++)
        right.add(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	0.073	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	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	0.073	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 Gruber¹ and Markus Holzer² and Oliver Ruepp²

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 $O(n \cdot n!)$ 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 $O(n)$ by not using comparisons.
 - Used by old IBM sorting machines.
- Bead Sort
 - Ingenious sorting algorithm for integers.
 - Uses gravity... how cool is that?